Center for Sustainable Science and Engineering Innovation

CSSEI

ARTICLE

An Efficient Task-Graph Scheduler for ML Pipelines with Data Locality, GPU Affinity, and Preemption Support

Naimul Hasan[1] and Tahmid Rahman[2]

[1]Department of Business Administration, Eastern Delta College of Management, 78 Sheikh Fazlul Haque Moni Avenue, Khulna, Bangladesh

[2]Department of Management Information Systems, Dhaka School of Business Technology, 23 Airport Road, Uttara Sector 5, Dhaka, Bangladesh

Abstract

Modern machine learning pipelines increasingly rely on heterogeneous task graphs that mix data preprocessing, feature engineering, model training, evaluation, and serving-oriented transformations. These pipelines exhibit nontrivial interactions between data locality, accelerator topology, and runtime variability, particularly when multiple workflows contend for shared GPU resources. A scheduler that ignores locality may overuse remote storage paths or interconnect links, while a scheduler that ignores GPU affinity may incur avoidable device-to-device transfers and kernel launch serialization. At the same time, long-running GPU-bound stages can reduce cluster responsiveness unless preemption is supported in a manner that respects the limited preemptibility of GPU execution and the high cost of state migration. This paper presents a task-graph scheduling approach for ML pipelines that jointly targets data locality, GPU affinity, and practical preemption. The design models each task's data footprint, read and write dependencies, and preferred device neighborhoods, while representing accelerator interconnects and storage tiers as a weighted topology graph. Scheduling decisions combine criticality-aware ordering with topology-aware placement and an explicit accounting of preemption overhead. Preemption is realized at safe boundaries with lightweight checkpointing and bounded rollback, enabling latency-sensitive jobs to obtain service without destabilizing throughput-oriented workloads. The proposed framework is described as a set of algorithms and runtime mechanisms that can be implemented over common cluster managers and GPU runtimes. Analytical considerations and experimental methodology are provided to characterize the scheduler's efficiency, fairness, and sensitivity to pipeline structure and contention.

1. Introduction

Task-graph execution is a common abstraction for end-to-end machine learning pipelines [1]. A pipeline can be expressed as a directed acyclic graph in which vertices represent computational tasks, such as decoding and augmenting samples, shuffling and batching, computing embeddings, executing forward and backward passes, applying optimizer steps, and materializing intermediate artifacts. Edges represent data dependencies that enforce precedence and describe the movement of intermediate results. In many production environments, pipelines are not executed as a single monolithic training job but as compositions of data services, transformation stages, and training components that interact through caches, feature stores, and shared datasets. These compositions lead to graphs with heterogeneous task sizes, varying degrees of parallelism, and a mixture of CPU- and GPU-dominated stages.

The performance of such graphs is shaped not only by the computational capacity of GPUs

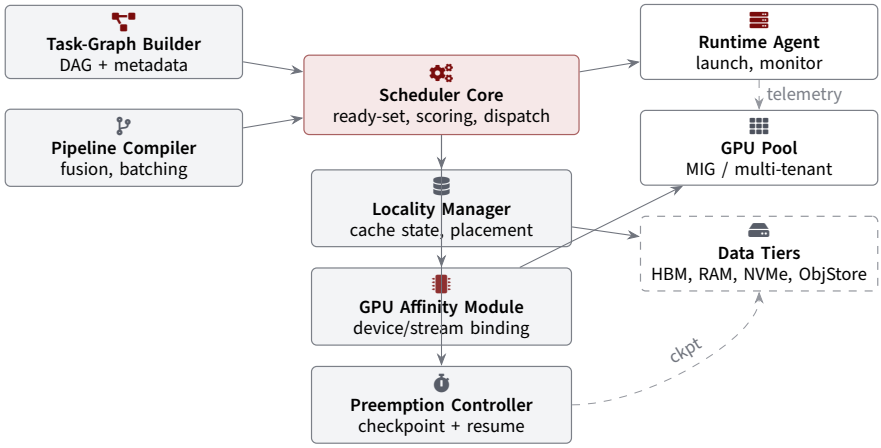

Figure 1. System-level architecture of a task-graph scheduler for ML pipelines: a central scoring/dispatch core integrates data-locality state, GPU affinity binding, and preemption (checkpoint/resume) while interacting with a runtime agent and multi-tier storage.

Table 1. Summary of ML task-graph workloads used in the evaluation.

Pipeline	#Tasks	Critical path (s)	Total data (GB)	Avg task time (ms)
Image classification (ResNet-50)	430	12.3	1.2	28.4
Language modeling (BERT)	680	18.7	2.8	32.1
Recommender system	510	9.4	3.6	21.7
Speech recognition	590	15.2	4.1	25.9
Multimodal vision+NLP	720	21.5	5.0	35.3

and CPUs but also by data movement costs [2]. A scheduler that is indifferent to locality may repeatedly fetch training shards from remote storage even when replicas are available on a nearby node, or may place adjacent tasks on devices that require expensive cross-host transfers. Similarly, modern GPU clusters are not uniform. Devices are arranged in topologies with nonuniform bandwidth and latency, including intra-node interconnects such as NVLink or PCIe switches and inter-node fabrics such as InfiniBand or Ethernet. Many ML stages exhibit affinity patterns that favor co-location on GPUs with high-bandwidth connectivity, especially for pipeline-parallel training, tensor-parallel collectives, gradient reductions, and stages that reuse cached activations or embeddings. When affinity is ignored, the runtime may compensate via additional communication, but this compensation can be large enough to dominate step time and to reduce overall cluster utilization [3].

A third pressure arises from multi-tenant operation. Production clusters typically host multiple pipelines concurrently, mixing latency-sensitive runs, small experiments, scheduled retraining, and throughput-oriented large jobs. In such settings, preemption is an important tool for maintaining responsiveness and enforcing service policies. However, GPU execution complicates preemption. Many kernels are not preemptible in a fine-grained sense, device memory is not cheap to snapshot, and ML training states can be large [4]. Aggressive preemption can therefore degrade throughput through repeated warm-up, cache invalidation, and data re-reading, while overly conservative preemption can starve short jobs and increase tail latency. The practical goal is not to preempt at arbitrary instruction boundaries but to offer controlled interruption points that are compatible with ML frameworks and that have predictable overhead.

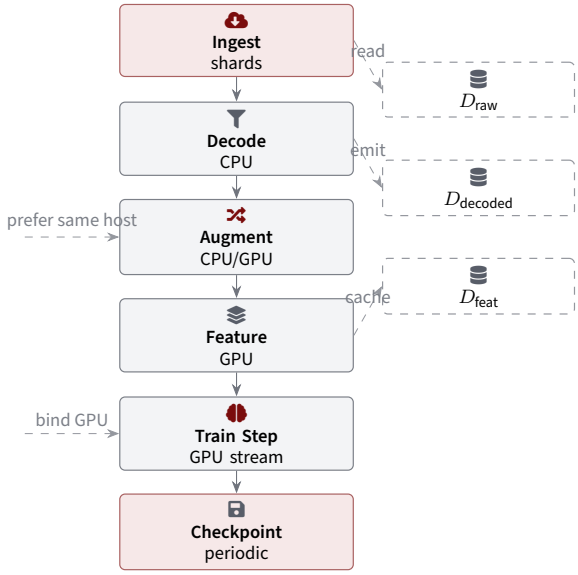

Figure 2. Representative ML pipeline task graph (DAG) annotated with lightweight locality and affinity hints: upstream CPU stages generate cached intermediates, while GPU stages prefer device-resident features and stable stream/device binding across iterations.

Table 2. Hardware configurations of the heterogeneous cluster.

Node type	CPU cores	GPUs / node	Interconnect
CPU-only	32	0	25 GbE
GPU-Small	32	2	PCIe 4.0, 100 GbE
GPU-Medium	64	4	NVLink 3, 200 GbE
GPU-Large	64	8	NVLink 4, 200 GbE

The problem addressed in this paper is the design of a task-graph scheduler that explicitly captures data locality, GPU affinity, and preemption cost within a unified decision process. The target is not a single static schedule but a runtime policy that can adapt to variability in task durations, data availability, and contention. The scheduler must respect dependency constraints, meet resource constraints, and remain robust when task execution times deviate from estimates [5]. It must also produce decisions quickly, since scheduling overhead can otherwise offset gains, particularly for graphs containing many short tasks.

The approach presented here centers on a topology-aware cost model, a criticality- and locality-sensitive dispatch policy, and a preemption mechanism that operates at safe boundaries with bounded state capture. The resulting system is intended to be compatible with common execution engines that already represent ML pipelines as graphs, including operators in dataflow systems and stages in training frameworks. The scheduler is described at an algorithmic level along with runtime mechanisms for state tracking, device placement, and preemption coordination.

Several principles guide the design [6]. Locality is treated as an explicit cost dimension rather than as an afterthought implemented through caching alone. GPU affinity is represented

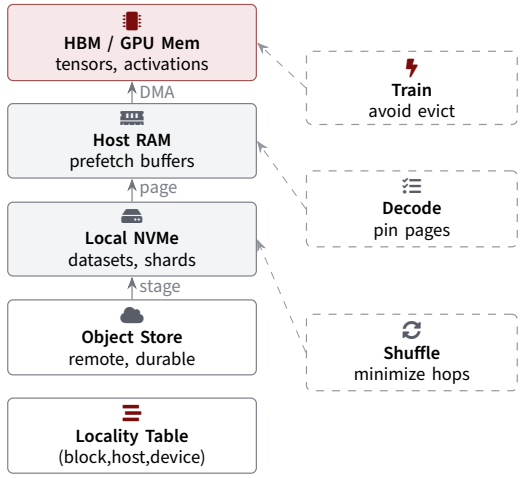

Figure 3. Data-locality model across storage tiers: the scheduler reasons about where blocks reside (device, host, local disk, remote store) and promotes high-reuse intermediates upward to reduce transfer and eviction overheads.

Table 3. Key scheduler configuration knobs and default settings.

Parameter	Description	Default value
α_{local}	Weight of data-locality score	0.5
α_{gpu}	Weight of GPU affinity score	0.3
α_{prio}	Weight of job priority	0.2
Preemption quantum (ms)	Minimum time slice before preempt	50
Warmup window (tasks)	Samples used for runtime estimation	64

using a graph of device neighborhoods, enabling the scheduler to prefer placements that reduce cross-neighborhood transfers for communication-heavy edges. Preemption is modeled as a decision with explicit overhead and is used selectively, guided by slack and fairness constraints rather than as a universal mechanism. The intent is to describe a scheduler that can improve throughput and predictability under realistic contention without relying on assumptions that are rarely satisfied in production, such as perfectly known task durations or uniform communication costs.

The remainder of the paper formalizes the system model and optimization objectives, then describes the scheduler architecture and algorithms, and finally discusses preemption mechanisms and evaluation [7]. Throughout, the focus is on the interaction between graph structure, data movement, and device topology, since these interactions are central to ML pipeline performance.

2. Problem Formulation and System Model

A machine learning pipeline is modeled as a directed acyclic graph $G = (V, E)$. Each vertex $v \in V$ denotes a task. A task can represent a data transformation, a training step, a parameter update stage, an evaluation run, or a materialization step that persists data for later consumption. An edge $e = (u, v) \in E$ indicates that v cannot begin until u completes and produces an output consumed by v [8]. For simplicity, the discussion assumes acyclicity,

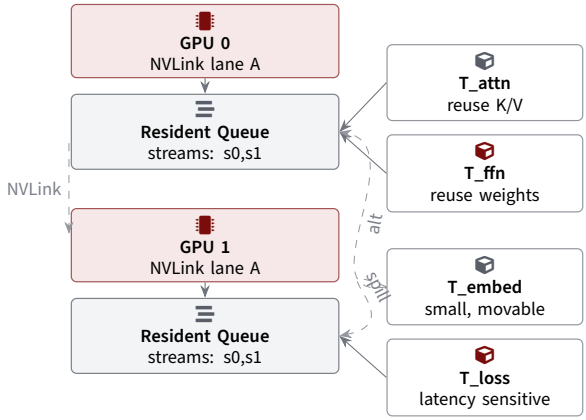

Figure 4. GPU affinity sketch: tasks with device-resident reuse are preferentially queued on the GPU where their hot tensors already live, while movable tasks may spill or migrate when contention or memory pressure increases.

Table 4. Impact of data-locality policies on communication and performance.

Policy	Local hit rate (%)	Remote traffic (GB)	Speedup vs. FIFO (x)
FIFO (no locality)	52.1	187.4	1.00
Locality-aware	78.6	92.3	1.21
Aggressive pinning	84.3	71.9	1.26
Topology-aware	88.9	63.5	1.32
Oracle (upper bound)	93.7	48.2	1.38

which matches many pipelines after unrolling iterations or treating iterative training as a sequence of repeated task instances. Where iterations are present, a common representation is a template graph executed repeatedly, potentially with loop-carried state, and the scheduler can operate over a window of forthcoming instances.

Each task v has a resource demand vector that includes CPU cores, GPU devices, memory, and possibly specialized accelerators. The focus here is on tasks that require one GPU for their dominant phase, although the model accommodates tasks that require multiple GPUs via gang semantics. For a task v, let r_v denote its GPU requirement, with $r_v \in \{0, 1, \dots\}$. A task with $r_v = 0$ is CPU-bound and is scheduled on CPU resources, but it can still influence GPU placement indirectly through data locality, since it may produce data used by GPU tasks [9].

The cluster is represented as a set of nodes N, each with a set of GPUs $D(n)$ and CPU resources. The GPUs across the cluster form a topology graph H whose vertices are devices and whose edges represent connectivity, annotated with bandwidth and latency. This topology graph is not purely physical, because it can incorporate effective bandwidth under contention and can include logical edges representing communication paths through switches or network fabric. For a pair of devices i and j, an effective transfer cost function $c_{\text{dev}}(i, j, s)$ gives the expected cost to transfer s bytes between them, capturing both latency and bandwidth as well as potential overheads from protocol and synchronization.

Data objects in the pipeline include input shards, intermediate tensors, cached features, and checkpoints. Let $\boldsymbol{O}$ be the set of objects. Each edge (u, v) is associated with a set of objects

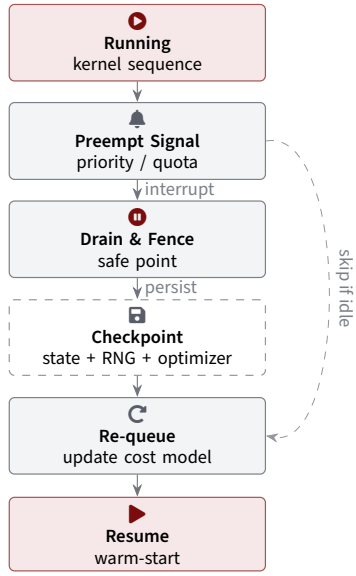

Figure 5. Preemption state machine: upon a preempt signal, the runtime drains to a safe point, checkpoints task state, and re-queues it for later warm-start resumption under updated priorities and resource availability.

Table 5. Effect of GPU affinity strategies on utilization and throughput.

GPU placement strategy	GPU utilization (%)	PCIe bandwidth (GB/s)	Throughput (samples/s)
Random	61.4	19.2	12.7k
Round-robin	68.9	17.5	13.9k
Device-local-first	77.3	14.8	15.4k
Topology-aware packing	81.6	13.9	16.1k
Affinity+load balance	84.2	13.7	16.5k

produced by u and consumed by v [10]. The scheduler needs a model of where objects are located. For each object o, let $L(o)$ be the set of locations where o is available. Locations can be node-local storage, distributed caches, memory-mapped files on a host, or device memory on a particular GPU. Since device memory is limited, many objects are not persistently resident on GPUs, but intermediate tensors may be reused within small neighborhoods, such as across fused operators or within a pipeline-parallel stage. The model allows $L(o)$ to include device-level placements when relevant [11].

A task v has an input set $I(v) \subseteq \mathcal{O}$ and an output set $O(v) \subseteq \mathcal{O}$. The total data movement cost for executing v on device d depends on where inputs are located and where outputs will be consumed. Because outputs are not yet placed at the time of scheduling v, the scheduler uses estimates derived from downstream dependencies, locality policies, and the expected consumer devices. This introduces a planning problem that mixes immediate placement with anticipation of future transfers.

Execution time is uncertain. For each task v, an estimate $\hat{t}_v(d)$ denotes the expected runtime on device d, accounting for device type and possibly for co-resident load. The actual runtime $t_v(d)$ may differ [12]. The scheduler maintains online estimates via profiling and exponential smoothing, and it can adjust for distributional characteristics such as heavy tails. The formulation here treats $\hat{t}_v(d)$ as an input and discusses robustness through policy design rather

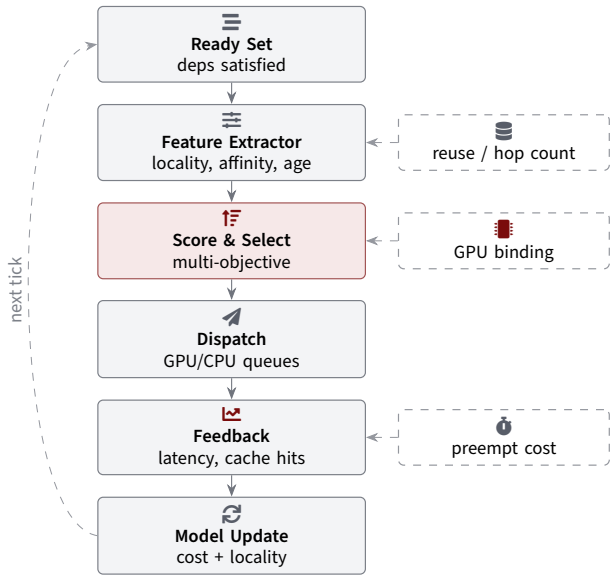

Figure 6. Online scheduling loop: ready tasks are featurized using locality/affinity signals, ranked by a multi-objective score, dispatched to execution queues, and fed back into an evolving cost/locality model for subsequent decisions.

Table 6. Preemption overhead and deadline behavior under different policies.

Scenario	Preemption latency (ms)	Wasted work (%)	Deadline misses (%)
No preemption	0.0	0.0	7.4
Naïve kernel-level preempt	23.8	12.6	4.3
Task-boundary preempt	11.5	6.1	3.2
Predictive preempt	8.2	4.7	2.8
Predictive + locality-aware	8.9	5.0	2.3

than requiring perfect estimates.

The precedence constraints are standard. A task v becomes ready when all predecessors complete and its inputs are materialized at some locations. However, readiness is nuanced when data availability is delayed by transfer. A task can be logically ready by dependency completion but physically blocked by input transfers [13]. The scheduler therefore maintains a readiness state that includes both dependency satisfaction and data residency.

The scheduling objective in a multi-tenant environment typically includes throughput, mean completion time, tail latency, and fairness. A single scalar objective is often insufficient. The scheduler described here uses a weighted composite objective that can be tuned according to service policy. The key novelty is that data locality and GPU affinity are treated as first-class terms, and preemption is explicitly priced rather than being an implicit side effect [14].

Let π denote a policy that assigns each task instance v to a device d and a start time, subject to resource constraints. The makespan of a graph instance is the completion time of its sink tasks. In a multi-graph setting, each job J has a graph G_J and a release time. The system seeks a policy that achieves stable utilization and acceptable latency across jobs. A simplified formulation for a single job focuses on minimizing a weighted sum of completion time and data movement costs [15]. For multiple jobs, the formulation introduces priorities and fairness

Table 7. Comparison of the proposed scheduler against baseline policies.

Scheduler	Norm. JCT (lower is better)	GPU utilization (%)	Jain fairness index
FIFO per-GPU	1.00	63.7	0.71
Fair-share (DRF)	0.89	66.4	0.84
Gang scheduling	0.94	69.1	0.79
Graph-aware (no preempt)	0.82	72.8	0.86
Proposed scheduler	0.73	79.5	0.91

Table 8. Scalability of the scheduler under increasing cluster and workload sizes.

Nodes	Concurrent pipelines	Throughput (jobs/hour)	Scheduler CPU overhead (%)
4	4	92	3.1
8	8	191	3.9
16	16	387	4.4
32	24	711	4.9
64	32	1,326	5.6

constraints.

A useful abstraction is to view scheduling as a sequence of dispatch decisions from a ready set. At time τ, a set of tasks $R(\tau)$ is eligible to run given dependencies and data movement progress. The scheduler selects tasks from $R(\tau)$ to assign to available devices, potentially triggering data transfers. If preemption is enabled, the scheduler may also interrupt running tasks at safe points, releasing devices for higher priority work. The policy must choose when to preempt and which tasks to resume later [16].

The remainder of this paper develops a cost model for locality and affinity, then describes a scheduler that approximates the multi-objective problem using online heuristics with explicit pricing of data transfers and preemption overhead. The emphasis is on decisions that can be made efficiently while still capturing the dominant costs in ML pipeline execution.

3. Locality and Affinity Cost Modeling

Data locality in ML pipelines spans multiple tiers. Input datasets may reside in remote object storage, in distributed file systems, or in node-local caches. Intermediate results can be materialized in host memory, in local SSDs, or in distributed caches [17]. Some intermediates are transient and only relevant within short windows, while others are reused across epochs or across jobs. A locality-aware scheduler benefits from a model that estimates the incremental cost of executing a task on a given device due to data movement across tiers and across the accelerator topology.

For a task v scheduled on device d, define its input transfer cost as a sum over required objects. If an object $o \in I(v)$ is available at a location that is co-located with d, the transfer cost may be small. If o must be fetched from a remote node, the cost includes network transfer and possibly decompression or deserialization [18]. Let s_o be the size of object o. Let $\ell(d)$ denote the host node on which device d resides. For each object o, define a best-available source location relative to d, chosen to minimize transfer cost subject to availability and policy constraints. The input cost can be approximated as

$$C_{\text{in}}(v, d) = \sum_{o \in I(v)} \min_{\lambda \in L(o)} c_{\text{loc}}(\lambda, d, s_o) \quad , \tag{1}$$

Table 9. Observed queueing behavior across priority classes under mixed workloads.

Priority class	Share of tasks (%)	Avg wait time (ms)	Preemptions / 1000 tasks
High (latency-sensitive)	18.2	9.7	42
Medium	46.5	21.3	17
Low (batch)	35.3	48.9	6
Best-effort background	0.8	112.4	2

where $c_{\mathrm{loc}}(\lambda, d, s)$ is a tier- and topology-aware transfer cost from location λ to device d. If λ is a node-local SSD on $\ell(d)$, c_{loc} may reflect local I/O throughput and CPU overhead. If λ is a remote node, c_{loc} includes network cost. If λ is another device d', the cost can be mapped through $c_{\mathrm{dev}}(d', d, s)$.

The output transfer cost depends on consumers [19]. If a produced object is consumed by tasks likely to be scheduled on certain devices or neighborhoods, placing the producer accordingly can reduce future costs. Exact anticipation requires global planning, but an online scheduler can approximate by using expected consumer neighborhoods, which can be inferred from the graph structure and prior placements. For an output object $o \in O(v)$, let $\Gamma(o)$ be the set of immediate consumer tasks. The scheduler maintains an estimated distribution over where consumers will run, represented as a set of candidate devices or neighborhoods with probabilities derived from current queue states and affinity constraints. Denote this distribution by $P_o(d')$. The expected output cost is then [20]

$$C_{\mathrm{out}}(v, d) = \sum_{o \in O(v)} \sum_{d'} P_o(d')\, c_{\mathrm{loc}}(d, d', s_o) \quad . \tag{2}$$

In practice, the scheduler may collapse P_o to a small set of representative neighborhoods, such as GPUs on the same host, GPUs connected by high-bandwidth links, or a designated parameter-server location.

GPU affinity extends locality by encoding preferences that are not purely data-size driven. Many ML operations benefit from placing related tasks on GPUs with strong connectivity even when the transferred bytes are modest, because synchronization and collective operations can have latency-sensitive phases. Affinity can also reflect reuse of ephemeral states, such as cached kernels, autotuning results, or warmed-up execution contexts. Additionally, for pipeline-parallel training, consecutive stages may prefer fixed GPU assignments to maintain steady-state flow [21].

Affinity is modeled as a set of soft constraints that add penalties when violated. For tasks u and v with a relationship that benefits from proximity, an affinity weight $a_{u,v} \geq 0$ is defined. The relationship may correspond to an edge in the pipeline graph or to a higher-level semantic coupling, such as tasks that share model parameters or that participate in a collective. For device assignments d_u and d_v, define a topology distance $\delta(d_u, d_v)$ derived from the effective transfer cost for a unit message or from a normalized latency measure. The affinity penalty can be expressed as

$$C_{\mathrm{aff}}(u, v, d_u, d_v) = a_{u,v}\, \phi\big(\delta(d_u, d_v)\big) \quad , \tag{3}$$

where ϕ is a monotone function that maps distance to penalty [22]. A common choice is a convex function that penalizes crossing certain boundaries more than small intra-boundary shifts, reflecting the fact that crossing a host boundary or a fabric boundary is often much more expensive than moving within a host. The scheduler can precompute neighborhoods

using clustering on H, such as grouping GPUs under the same PCIe switch or within the same NVLink island.

Data locality and affinity interact because a placement that is ideal for input locality may be poor for affinity and vice versa. A task might prefer a GPU near a cached dataset shard but also prefer a GPU close to downstream training stages. The scheduler therefore uses a combined cost [23]. For a task v assigned to d, define an estimated incremental cost

$$C(v, d) = \alpha\, C_{\text{in}}(v, d) \quad + \quad \beta\, C_{\text{out}}(v, d) \quad + \quad \gamma\, C_{\text{aff}}(v, d) \quad , \tag{4}$$

where $C_{\text{aff}}(v, d)$ aggregates affinity penalties between v and a set of related tasks with known or hypothesized placements. The weights α, β, γ are policy parameters. While a single set of weights may not cover all workloads, the framework permits tuning by job class. Latency-sensitive inference pipelines may set α and γ higher to reduce transfer and synchronization delays, while throughput-oriented training may emphasize stability and reduce β to avoid overreacting to speculative downstream placements [24].

Preemption cost is also modeled explicitly because it interacts with locality and affinity. Preempting a task can discard locality benefits if cached data is evicted or if partially completed work must be recomputed. It can also disrupt affinity by forcing migration to a different device when the task resumes. Let $p_v(d)$ denote the expected cost of preempting task v while it runs on device d. This cost depends on the existence of safe preemption points, the frequency of checkpointing, and the size of state to be saved [25]. If preemption is only permitted at operator boundaries, then $p_v(d)$ may include a bounded waiting time until the next boundary plus the time to serialize state. An approximate cost model is

$$p_v(d) = \mathbb{E}\big[w_v(d)\big] \quad + \quad c_{\text{ckpt}}(v, d) \quad + \quad c_{\text{warm}}(v, d) \quad , \tag{5}$$

where $w_v(d)$ is the wait until a safe point, c_{ckpt} is the time to capture minimal state, and c_{warm} accounts for resumed execution overhead such as cache rebuilds or re-reading data. This explicit pricing allows the scheduler to compare the benefit of serving a higher priority task against the disruption cost incurred by preempting a running one.

A key modeling decision is how to represent contention [26]. Transfer costs and runtimes depend on concurrent activity. A tractable approximation treats each resource class as having an effective capacity that can be estimated from recent measurements, and uses this capacity to scale cost terms. For example, if the network is saturated, c_{loc} increases for remote transfers. Similarly, if a host's CPU is congested, local preprocessing tasks may slow down, indirectly delaying GPU tasks that depend on them. The scheduler maintains moving averages of throughput and latency for each tier and for key topology edges, updating costs accordingly. While this approach does not capture all interactions, it provides a feedback loop that can stabilize decisions under changing load [27].

The cost model supports a scheduling rule that chooses placements by minimizing a composite score that includes expected runtime, locality cost, affinity penalty, and optionally preemption cost if preemption is invoked. This rule is used in conjunction with graph-criticality measures so that the scheduler does not over-optimize local costs at the expense of delaying the critical path.

4. Scheduler Architecture and Core Algorithm

The scheduler operates as an online system that repeatedly processes events such as task readiness, resource availability, completion notifications, and policy triggers for preemption. The

internal state includes the task graph structure, runtime estimates, data location metadata, and device topology metrics. Decisions are made at two levels [28]. A global component performs topology-aware placement across nodes and GPU neighborhoods. A local component manages dispatch queues within a node or within a neighborhood, enforcing GPU affinity and selecting among ready tasks.

The global component maintains a view of available devices and their neighborhood memberships. A neighborhood is a subset of GPUs with relatively high mutual bandwidth and low latency, such as GPUs connected by a high-bandwidth intra-node fabric. Neighborhoods can overlap in the physical sense, but the scheduler uses a partition for simplicity [29]. Each neighborhood has a capacity expressed in available GPUs and in auxiliary resources such as CPU threads for input pipelines. Jobs can declare neighborhood preferences, for example requiring that a gang of k GPUs for tensor parallelism must be selected from the same neighborhood to reduce all-reduce cost.

The scheduler uses a task ranking function that combines criticality and cost. Criticality is computed from the graph as an estimate of remaining work to completion. For a task v, define a downstream length estimate $L(v)$ as the maximum over all paths from v to a sink of the sum of estimated runtimes and expected transfer costs along the path. Let $S(v)$ denote the estimated start slack relative to a job-level target completion time [30]. In a multi-tenant system, slack can be derived from a target service curve or from priority weights. A task with low slack is more urgent.

To incorporate locality and affinity, the scheduler defines for each task v and candidate device d a placement score that approximates the marginal completion impact. A useful score is

$$\operatorname{score}(v,d) = \hat{t}_v(d) \quad + \quad C(v,d) \quad + \quad \eta\, Q(d) \quad - \quad \kappa\, \operatorname{crit}(v) \quad , \tag{6}$$

where $Q(d)$ is an estimate of queuing delay on device d, $\operatorname{crit}(v)$ is a normalized criticality measure derived from $L(v)$ and slack, and η, κ are weights that trade off load balancing against criticality. The sign convention reflects that higher criticality should reduce the score and thereby increase selection priority [31]. This score is evaluated over a restricted candidate set of devices rather than all devices, to keep decision latency low. Candidate selection is guided by data locality, selecting devices on nodes that already host input objects, and by affinity, selecting devices in neighborhoods where related tasks are or are expected to be.

The global placement algorithm proceeds as follows in a continuous event loop. When a task becomes eligible and at least one suitable device is available, the scheduler selects a device that minimizes the score. However, selecting tasks one by one can produce suboptimal outcomes when tasks require multiple GPUs or when co-scheduling benefits exist [32]. The scheduler therefore supports a batch placement mode for groups of related tasks, such as consecutive pipeline stages or a set of tasks participating in a collective. In batch mode, the scheduler solves a small assignment problem restricted to a candidate neighborhood set. The assignment can be approximated using greedy matching because candidate sets are small, and the primary goal is to avoid gross mismatches rather than to compute a globally optimal schedule.

A central challenge in task-graph scheduling is the tension between keeping devices busy and avoiding premature execution that increases future transfer costs. For example, executing a preprocessing task on a distant node may produce an output that must then be transferred to a GPU neighborhood for training, whereas delaying that preprocessing until a local CPU is available could avoid the transfer [33]. The scheduler addresses this by distinguishing between tasks that are locality-sensitive and tasks that are locality-agnostic. Locality sensitivity is

inferred from the ratio of estimated transfer cost to compute time. For task v, define

$$\rho_v(d) = \frac{C_{\text{in}}(v, d) + C_{\text{out}}(v, d)}{\hat{t}_v(d) + \epsilon} \quad , \tag{7}$$

with a small ϵ to avoid division by zero [34]. If $\rho_v(d)$ is high across candidates, then locality dominates, and the scheduler should be conservative in placement to avoid expensive transfers. If $\rho_v(d)$ is low, then compute dominates and keeping devices busy may be preferable. The scheduler uses ρ_v to adjust the weight η on queuing delay and the weight on transfer costs dynamically, so that locality-sensitive tasks are more strongly guided by data placement.

Affinity is maintained by introducing soft reservations. When a set of tasks forms a chain with high affinity, such as consecutive training stages, the scheduler may create an affinity group with a preferred neighborhood [35]. The group can be pinned to a neighborhood for a window of time to reduce churn. Pinning is not absolute, since it can conflict with preemption and fairness. Instead, the scheduler uses a hysteresis mechanism. Deviations from the preferred neighborhood are allowed but incur increasing penalties that decay over time. This reduces oscillations in which tasks repeatedly migrate between neighborhoods [36].

The local component manages per-device queues and enforces that tasks assigned to a device respect execution constraints. For GPU tasks, the local component also interfaces with stream priorities and concurrent kernel execution features, but the scheduler described here does not rely on fine-grained kernel-level preemption. Instead, it treats the task as the primary scheduling unit. Within a device, tasks can be executed in a cooperative manner if the runtime supports it, but the default assumption is exclusive occupancy for the dominant phase of each task to simplify modeling of runtimes and interference.

The algorithm also accounts for data transfer scheduling [37]. Transfers can be initiated proactively when a task is likely to be scheduled on a certain device. For an eligible task v, if the scheduler tentatively selects a device d, it can begin transferring inputs to $\ell(d)$ or directly to d while waiting for device availability, overlapping transfer with computation. This requires a commitment mechanism so that speculative transfers do not waste bandwidth. The scheduler uses a bounded speculation policy in which only a limited number of tasks per job may have speculative transfers in flight, and speculation is prioritized for tasks on the critical path. If a speculative placement is revoked due to preemption or updated priorities, the scheduler can reuse transferred data if it remains useful in the selected neighborhood, or it can cancel remaining transfer if supported by the transport [38].

To ensure fairness and avoid starvation, especially when locality and affinity create strong preferences for certain neighborhoods, the scheduler introduces a fairness term that penalizes allocating repeated service to the same job beyond its share. Let $f_J(\tau)$ represent the accumulated service for job J up to time τ, measured in GPU-seconds adjusted by device performance. A fair-share target $g_J(\tau)$ can be derived from weights. The scheduler adds a penalty proportional to $f_J(\tau) - g_J(\tau)$ when selecting among tasks from different jobs. This penalty is applied at the job level and then distributed across tasks [39]. The effect is that a job that has already received more than its share will see increased scores, making its tasks less likely to be selected when contention exists, unless they are critical for meeting deadlines or have high preemption priority.

The computational complexity of the online decision process is controlled by restricting candidates and by maintaining incremental updates of criticality metrics. The downstream length $L(v)$ can be precomputed for a static graph using a reverse topological traversal with edge costs included. For repeated execution of a template graph, $L(v)$ can be reused. When

dynamic edges or data-dependent branches exist, the scheduler can compute $L(v)$ lazily for affected subgraphs [40]. Candidate device selection uses locality hints, such as nodes that host the majority of required inputs, and neighborhood affinity hints, limiting the evaluation of score(v, d) to a small set. As a result, dispatch latency can remain small even for graphs with many tasks.

A practical concern is that task boundaries in ML pipelines can be fine-grained, especially in data preprocessing. Scheduling at too fine a granularity increases overhead. The scheduler therefore supports task coalescing as a preprocessing step, where multiple small operators are fused into a larger schedulable unit when they have strong locality and when the fused runtime remains predictable. Coalescing reduces the number of scheduling events and increases the opportunity to amortize data movement [41]. Importantly, coalescing must respect preemption needs, since fusing too aggressively can create long tasks that are difficult to interrupt. The scheduler uses a coalescing policy that limits fused task duration to a configurable bound, thereby preserving manageable preemption points.

The core algorithm can be summarized as an online policy: compute readiness with data-aware states, rank tasks by criticality and slack, select candidate neighborhoods based on locality and affinity, choose device assignments minimizing a composite score with fairness adjustment, initiate bounded speculative transfers, and dispatch tasks to local queues. Preemption is invoked through a separate mechanism described next, but the core algorithm provides the information needed to decide whether preemption is beneficial by estimating the opportunity cost of waiting versus the cost of interruption.

5. Preemption Protocol and Runtime Mechanisms

Preemption support in GPU-centric ML pipelines requires a careful definition of what it means to interrupt a task [42]. Fine-grained kernel preemption is not uniformly available across devices and runtimes, and even when supported, it may introduce overheads or constraints that complicate portability. A more practical approach is cooperative preemption at safe points. A safe point is a location in the task's execution where state can be captured with bounded overhead and where resumption can proceed without violating correctness or significantly increasing nondeterminism beyond acceptable bounds.

In the context of task graphs, a natural safe point is at task boundaries. If tasks represent coarse stages, such as a full training step including forward, backward, and optimizer update, then boundaries may be too coarse for responsiveness [43]. Conversely, if tasks are very fine-grained, boundaries may be plentiful but capturing state between tasks may still be expensive if intermediate tensors are large. The scheduler described here uses a tiered safe-point model that allows additional boundaries within a task when the runtime supports it. For example, a training step can be decomposed into sub-tasks corresponding to forward pass, backward pass, gradient reduction, and optimizer update, with dependencies between them. These sub-tasks can be represented explicitly in the task graph, enabling preemption between them. This representation increases graph size but provides more opportunities for interruption without requiring kernel-level preemption [44].

When interrupting a running task v on device d, the scheduler coordinates with the runtime to reach the next safe point. Let b_v denote the average time between safe points for v. If safe points are only at task completion, b_v is the task runtime. If the task contains internal safe points, b_v can be smaller. A preemption request issued at time τ incurs an expected waiting time $w_v(d)$ until the next safe point. If safe points are periodic and independent of τ, an approximation is $\mathbb{E}[w_v(d)] \approx b_v/2$. In practice, the runtime can report progress to reduce

uncertainty [45].

State capture at a safe point is task-specific. For data preprocessing tasks, state may be small and capturing it may be unnecessary, since restarting the task is cheap relative to transfer and compute. For training tasks, state includes model parameters, optimizer state, and potentially random number generator state to ensure reproducibility. Capturing full model state at every preemption is expensive, so the design uses incremental checkpointing. The runtime periodically creates checkpoints of training state at a configurable interval, such as every few steps, and a preemption can roll back to the latest checkpoint [46]. The preemption overhead then includes the loss of work since the last checkpoint, which is bounded by the checkpoint interval. If the interval is chosen to bound loss to a small fraction of step time, such as 2% to 10%, then preemption becomes more tractable, at the cost of extra checkpoint I/O.

To formalize this trade-off, let T be the expected runtime of a training step task, and let checkpoints occur every k steps. The expected recomputation after preemption is approximately $(k-1)\,T/2$ if preemption time is uniformly distributed across the interval. Let C_{save} be the time to save a checkpoint and C_{load} the time to load it. The amortized overhead per step due to checkpointing is C_{save}/k, while the expected overhead due to preemption events depends on their rate. If preemption events occur with rate λ per step, the expected additional overhead per step from recomputation and reload is $\lambda\,((k-1)\,T/2 + C_{\text{load}})$. The scheduler can choose k based on an acceptable overhead budget and an expected preemption rate [47]. While λ is not known a priori, it can be estimated from observed contention and policy.

The scheduler integrates preemption decisions into its objective by treating preemption as an option when a high-priority task cannot be scheduled otherwise. Suppose a ready high-priority task h requires a GPU and no suitable GPU is available. The scheduler considers preempting a running task v on some GPU d. The benefit of preemption is the reduced waiting time for h and potentially improved deadline adherence [48]. The cost is $p_v(d)$, including waiting to safe point, checkpoint or rollback overhead, and locality loss. The decision rule can be expressed as comparing an urgency value $U(h)$ against the minimum preemption cost among candidates. One form is

$$\text{preempt}(v, d \to h) \quad \text{if} \quad U(h) \;>\; p_v(d) \;+\; \Delta_{\text{local}}(v, d) \quad , \tag{8}$$

where $\Delta_{\text{local}}(v, d)$ accounts for the additional locality and affinity penalties incurred by migrating v upon resumption. $U(h)$ can be derived from slack, priority class, and queueing delay. This comparison is performed over a subset of running tasks, prioritizing those with low preemption cost, such as tasks near a safe point or tasks with small state [49].

A difficulty is that preempting a task may not immediately free the GPU if it must wait until a boundary. To avoid excessive delays, the scheduler issues preemption requests with a lead time. If an urgent task is expected to become ready soon, the scheduler can prepare by requesting a safe-point stop for a running task so that a GPU becomes available at approximately the right time. This anticipatory preemption reduces the chance that an urgent task waits behind long-running kernels. However, anticipation can waste work if the urgent task does not materialize [50]. The scheduler therefore limits anticipation to cases with high confidence, such as when the urgent task's predecessors are near completion.

Resuming preempted tasks raises placement questions. If the original GPU is still available and retains relevant cached data, resuming there preserves locality and affinity. If not, the scheduler may migrate the task. Migration cost depends on the amount of state and on

data locality [51]. For training tasks, parameters may be stored in host memory or in a distributed parameter store, making migration feasible. For tasks with large device-resident state, migration can be expensive. The scheduler maintains an affinity for the original device and neighborhood for preempted tasks, represented as an increased affinity weight that decays as time passes. This encourages resumption on the same neighborhood when feasible but permits migration when necessary for fairness or capacity.

Preemption also affects correctness and determinism [52]. Some ML pipelines tolerate nondeterministic execution order and minor variations due to floating point reductions, while others require reproducibility. The scheduler can enforce determinism constraints by restricting preemption points to locations that do not change operation ordering in a way that affects results. For example, if a task uses nondeterministic GPU kernels, restarting it can change outcomes. In such cases, the runtime can either enforce deterministic kernels or annotate tasks as non-preemptible except at specific boundaries with consistent replay. The scheduler respects these annotations by setting preemption cost to infinity for non-preemptible regions [53].

Runtime integration requires instrumentation. The execution engine must report task start, progress, and completion events, as well as data materialization and location updates. For safe-point preemption, tasks must periodically check for stop requests and reach a consistent boundary. In GPU execution, this is often implemented by structuring work into a sequence of kernels and synchronization points, allowing the runtime to stop between sequences. For training, checkpoints can be implemented using framework primitives that serialize model and optimizer state [54]. The scheduler does not assume a specific framework, but it requires that checkpoint creation and restore are available with bounded time, and that tasks can be restarted from a checkpoint without violating dependencies.

A related concern is memory pressure. Preemption can increase memory usage if checkpoints are buffered or if partially computed outputs are retained. The scheduler therefore treats memory as a constrained resource. When deciding to preempt, it checks whether the system has capacity to store any required checkpoint and whether resuming later would exceed memory limits [55]. The scheduler can also trigger eviction of cached objects to make room, but eviction reduces locality benefits, so the decision is coupled to the locality model. In general, the system prefers to store checkpoints in a tier with predictable performance, such as local SSD, and to use compression where feasible, recognizing that compression consumes CPU and can contend with preprocessing.

The preemption protocol is designed to be compositional with the core scheduling algorithm. The core algorithm assigns tasks and initiates transfers. The preemption mechanism can override assignments by stopping tasks and re-queuing them [56]. The scheduler maintains consistent state by treating preempted tasks as incomplete and by invalidating speculative transfers that are no longer aligned with expected placements. To avoid thrashing, the scheduler applies rate limits and cool-down windows, ensuring that the same task is not preempted repeatedly within a short time unless policy requires it. Cool-down can be expressed as a minimum run time after resumption before a task becomes eligible for preemption again, unless it belongs to a low-priority class.

The combined system aims to provide responsiveness to urgent tasks while keeping throughput degradation controlled through explicit preemption cost modeling, bounded checkpoint intervals, and locality-preserving resumption preferences.

6. Evaluation Methodology and Results

Evaluating a task-graph scheduler with locality, affinity, and preemption requires capturing a range of pipeline structures and contention patterns [57]. The methodology described here focuses on measurable metrics, controlled experiments, and sensitivity analyses that isolate the contributions of locality-aware placement, affinity-aware neighborhood selection, and preemption decisions. The evaluation can be conducted using a prototype integrated with a task-graph execution engine, or using a trace-driven simulator parameterized by measured transfer and compute costs. A prototype is preferable for validating overheads and integration complexity, while simulation enables broader sweeps across workloads and cluster scales.

Workloads are drawn from representative ML pipeline motifs. One motif is an input-heavy training pipeline where decoding and augmentation stages are CPU-bound but produce batches consumed by a GPU training stage, with optional caching of preprocessed samples [58]. Another motif is a multi-stage training pipeline with pipeline parallelism across GPUs, where consecutive stages exchange activations. A third motif is a feature-engineering pipeline with a mixture of join-like operations, embedding lookups, and GPU inference used for data labeling. A fourth motif is a multi-job environment combining short hyperparameter trials with a long-running training job, which stresses preemption and fairness. These motifs can be instantiated with varying degrees of parallelism and varying dataset shard placements across nodes.

The cluster topology is a critical variable. Experiments should include at least two topologies: a topology with strong intra-node connectivity, such as multiple GPUs per host with high-bandwidth links forming neighborhoods, and a topology with weaker intra-node links where PCIe dominates [59]. Inter-node connectivity should be parameterized to reflect different network fabrics. Data storage tiers can be modeled with remote object storage, distributed file system, and node-local cache, each with measured throughput and latency distributions. The scheduler's topology graph H and location cost functions c_{loc} are derived from these measurements.

The primary performance metrics include job completion time, mean and tail latency for latency-sensitive jobs, overall throughput measured in completed pipeline instances per unit time, and cluster utilization measured in GPU occupancy adjusted by device performance. Additional metrics quantify data movement, such as total bytes transferred across the network, bytes transferred across host boundaries, and bytes moved between GPUs. These metrics connect directly to locality and affinity [60]. For preemption, metrics include the number of preemption events, total recomputation time due to rollbacks, checkpoint overhead, and the distribution of waiting time to safe points.

Baselines for comparison include a topology-agnostic scheduler that uses a simple greedy policy based on earliest available device, a locality-only scheduler that places tasks based on input proximity but does not model GPU affinity, and a scheduler with priority-based preemption but without explicit preemption cost modeling. Another useful baseline is a policy that pins jobs to fixed GPU sets, which can preserve affinity but may harm fairness and responsiveness. The proposed scheduler can be compared to these baselines under identical workloads and resource configurations.

A key set of experiments isolates data locality [61]. In an input-heavy pipeline, nodes may cache different shards. A topology-agnostic scheduler may schedule GPU tasks on any available GPU, causing batches to be transferred from remote caches or remote storage. The locality-aware scheduler should preferentially place GPU tasks on nodes where the required

shards are present, reducing network traffic. Under moderate contention, this reduction can translate into lower step time variability because remote transfers introduce larger variance. The expected observation is that network bytes transferred per training step decreases and that GPU idle time due to input stalls decreases [62]. The magnitude depends on cache hit rates. When cache hit rate is high, such as 80% to 95%, locality-aware placement can avoid most remote transfers. When hit rate is low, the scheduler’s benefit is smaller, and the policy should avoid over-constraining placement that would leave GPUs idle.

Affinity evaluation focuses on communication-heavy pipelines. For pipeline parallel training, placing adjacent stages on GPUs with high-bandwidth connectivity can reduce activation transfer time and can stabilize pipeline bubbles [63]. If affinity is ignored, stages may be placed across host boundaries, leading to larger communication delays and reduced effective throughput. The affinity-aware scheduler should cluster stages within neighborhoods when capacity permits, and should prefer topologies with better collective performance for gang-scheduled tasks. The expected effect is a reduction in per-step communication overhead and a reduction in variance, especially when multiple jobs share the cluster and the scheduler must make nontrivial trade-offs. The affinity penalty function $\phi(\delta)$ influences how strongly the scheduler avoids crossing boundaries. Sensitivity analysis varies γ and the shape of ϕ to determine how robust performance is to mis-specification [64]. A policy that is too aggressive may overfit to affinity and cause load imbalance, while a policy that is too weak may approach the baseline.

Preemption evaluation examines responsiveness. In a mixed workload, a long-running training job occupies GPUs, and short, latency-sensitive evaluation jobs arrive intermittently. Without preemption, evaluation jobs may wait for long tasks to finish, increasing tail latency. With naive preemption, the training job may be interrupted frequently, increasing total completion time and reducing throughput [65]. The proposed cost-aware preemption aims to strike a balance by selecting preemption targets with low expected cost and by respecting cool-down windows. The evaluation measures the tail latency of short jobs, such as the 95% or 99% percentile, and the throughput impact on the long job. A useful result characterization is a Pareto curve showing latency improvements versus throughput loss as the urgency threshold $U(h)$ is varied. The scheduler’s ability to bound recomputation through checkpoint intervals can be evaluated by varying k and measuring the fraction of wasted work. For example, if checkpoints are taken every few steps, the expected wasted work can be kept below a policy target such as 5% of total compute, though the checkpoint overhead may add steady-state cost [66]. The appropriate setting depends on preemption frequency and storage performance.

Another important evaluation dimension is scheduling overhead. The proposed scheduler performs candidate selection and score evaluation. Overhead is measured as the time from a scheduling event to dispatch decision. For graphs with many tasks, overhead must remain small relative to task durations [67]. Task coalescing can reduce overhead but may reduce preemption granularity. Experiments can vary coalescing thresholds to measure the trade-off between overhead and responsiveness. A practical goal is that scheduling overhead remains a small fraction of task time for typical tasks, such as below 1% to 2% for GPU-heavy tasks, while acknowledging that preprocessing tasks may be shorter and thus more sensitive.

Robustness to runtime estimation error is evaluated by injecting noise into $\hat{t}_v(d)$ or by using workloads with high variability. Since the scheduler’s ranking and placement use estimates, errors can cause suboptimal placements. The policy’s reliance on transfer and topology costs can partially compensate because these costs can be measured more reliably than compute time, especially for stable interconnects [68]. However, transfer costs can also vary under contention. The evaluation therefore includes scenarios with changing network load, where

transfer costs increase dynamically. The scheduler's feedback loop updates c_{loc} based on observed throughput and can shift placements accordingly. The stability of this adaptation can be evaluated by observing whether the system oscillates between neighborhoods or whether it converges to stable placements.

Fairness is evaluated by running multiple jobs with different weights and measuring service received relative to targets. The presence of locality and affinity constraints can distort fairness if some jobs have stronger constraints that concentrate demand on certain neighborhoods [69]. The fairness penalty term should mitigate this by making the scheduler more likely to allocate alternative neighborhoods to over-served jobs when possible. Metrics include deviation from fair share over time windows and the frequency of starvation events. Preemption can also influence fairness, since high-priority jobs may repeatedly interrupt others. Cool-down windows and cost-aware selection aim to reduce pathological cases where one job is constantly displaced.

The overall results from such an evaluation are expected to show that locality awareness reduces unnecessary data movement and improves GPU utilization in input-bound pipelines, that affinity awareness reduces communication overhead in multi-stage GPU pipelines, and that cost-aware preemption improves tail latency for short jobs while keeping recomputation and throughput loss bounded [70]. The magnitude of improvements depends on topology heterogeneity and workload mix. In uniform clusters with centralized storage and low reuse, locality benefits may be limited. In clusters with strong locality opportunities, such as high cache hit rates and repeated dataset reuse, the benefits can be larger. In communication-heavy training, affinity can have a clear effect because cross-boundary transfers are costly. Preemption benefits depend on the arrival rate of urgent jobs and the checkpointing policy [71]. The evaluation methodology is therefore designed to map performance across these regimes rather than to present a single headline number, since scheduler behavior necessarily reflects trade-offs among objectives.

7. Conclusion

This paper presented an approach to scheduling ML pipeline task graphs that explicitly integrates data locality, GPU affinity, and practical preemption. The system model represents tasks with data dependencies and resource demands, and it represents the cluster as a topology graph capturing nonuniform device connectivity and storage tiers. A cost model was developed to estimate input and output transfer costs, to encode affinity penalties across device neighborhoods, and to price preemption overhead in terms of waiting for safe points, checkpoint or rollback costs, and locality loss. Building on these models, an online scheduler architecture was described that combines criticality-aware task ranking with topology-aware placement and fairness adjustments, while using bounded speculative transfers to overlap movement with computation [72].

Preemption was addressed through cooperative safe points rather than fine-grained kernel interruption, with tiered boundaries enabled by representing internal stages as tasks where appropriate. Incremental checkpointing provides bounded rollback cost, and preemption decisions are guided by explicit comparisons between urgency and expected preemption cost, complemented by rate limiting and locality-preserving resumption preferences. An evaluation methodology was outlined to quantify the impact of locality and affinity on data movement and communication overhead, and to characterize the responsiveness and throughput effects of cost-aware preemption under multi-tenant contention.

The central theme is that locality, topology, and interruption are interdependent in ML

pipelines. Treating locality and affinity as first-class quantities in scheduling decisions, and treating preemption as a priced action rather than a binary capability, provides a structured way to navigate the trade-offs that arise in heterogeneous clusters. The algorithms and mechanisms described here are intended to be implementable over common task-graph runtimes and cluster managers, with adjustable parameters that allow different service policies to be expressed without changing the underlying abstractions [73].

References

[1] Dawn E. Bouman. "Distributed Systems". In: Springer International Publishing, Sept. 20, 2018, pp. 1204–1205. DOI: 10.1007/978-3-319-57111-9_2138.

[2] Julius Voggesberger, Peter Reimann, Dennis Treder-Tschechlov, and Bernhard Mitschang. "Auto-CEn: AutoML for Classifier Ensembles - Diversity-Based Classifier Selection and Decision Fusion Optimization". In: *2025 IEEE 12th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, Oct. 9, 2025, pp. 1–10. DOI: 10.1109/dsaa65442.2025.11248022.

[3] Kamila Orynbekova, Shirali Kadyrov, Andrey Bogdanchikov, and Saidakmal Oktamov. "A novel recommender system for adapting single machine problems to distributed systems within MapReduce". In: *Bulletin of Electrical Engineering and Informatics* 14.1 (Feb. 1, 2025), pp. 687–695. DOI: 10.11591/eei.v14i1.8370.

[4] Vincenzo De Maio, Atakan Aral, and Ivona Brandic. "A Roadmap To Post-Moore Era for Distributed Systems". In: *Proceedings of the 2022 Workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*. ACM, July 25, 2022, pp. 30–34. DOI: 10.1145/3524053.3542747.

[5] Rahul Malik, Sangkyum Kim, Xin Jin, Chandrasekar Ramachandran, Jiawei Han, Indranil Gupta, and Klara Nahrstedt. "MLR-index: An index structure for fast and scalable similarity search in high dimensions". In: *International Conference on Scientific and Statistical Database Management*. Springer. 2009, pp. 167–184.

[6] Danil Temnikov and Roman Dubinin. "Application Of Ai for Enhancing the Performance of Distributed Systems". In: *The American Journal of Engineering and Technology* 7.7 (July 31, 2025), pp. 180–185. DOI: 10.37547/tajet/volume07issue07-17.

[7] Yishuang Hu and Yimin Bo. "Reliability on Distributed System Considering Transmission Lines". In: *2022 5th International Conference on Power and Energy Applications (ICPEA)*. IEEE, Nov. 18, 2022, pp. 459–463. DOI: 10.1109/icpea56363.2022.10052571.

[8] Z.R. Mayransaev and A. B. Chernyshev. "GENERALIZED CIRCULAR CRITERION FOR THE ABSOLUTESUSTAINABILITY OF DISTRIBUTED SYSTEMS". In: *IZVESTIYA SFedU. ENGINEERING SCIENCES* 2 (July 1, 2021), pp. 182–189. DOI: 10.18522/2311-3103-2021-2-182-189.

[9] Geoffrey Neumann, Paul Grace, Daniel Burns, and Mike Surridge. "Pseudonymization risk analysis in distributed systems". In: *Journal of Internet Services and Applications* 10.1 (Jan. 8, 2019), pp. 1–16. DOI: 10.1186/s13174-018-0098-z.

[10] Aditi Mishra. *Global Convergence in Asynchronous Distributed Systems*. Feb. 3, 2020.

[11] Mattia Zaccarini et al. "TELKA: Twin-Enhanced Learning for Kubernetes Applications". In: *2024 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, June 26, 2024, pp. 1–6. DOI: 10.1109/iscc61673.2024.10733736.

[12] Sebastian Böhm and Guido Wirtz. "Towards Orchestration of Cloud-Edge Architectures with Kubernetes". In: Jan. 1, 2022, pp. 207–230.

[13] Friedrich Solowjow, Arash Mehrjou, Bernhard Schölkopf, and Sebastian Trimpe. *Minimum Information Exchange in Distributed Systems*. May 24, 2018.

[14] Mozhdeh Farhadi, Daniele Miorandi, and Guillaume Pierre. *Blockchain enabled fog structure to provide data security in IoT applications*. Dec. 10, 2018.

[15] Andrea Fieschi, Pascal Hirmer, Sachin Agrawal, Christoph Stach, and Bernhard Mitschang. "HySAAD – A Hybrid Selection Approach for Anonymization by Design in the Automotive Domain". In: *2024 25th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, June 24, 2024, pp. 203–210. DOI: 10.1109/mdm61037.2024.00044.

[16] Karol Marszałek, Adam Domański, Rafał Cupek, and Marek Drewniak. "Testing Quality of Service of communication system for AGV fleet with Software-Defined Network". In: *2023 IEEE International Conference on Big Data (BigData)*. IEEE, Dec. 15, 2023, pp. 5078–5086. DOI: 10.1109/bigdata59044.2023.10386380.

[17] Judah M. Diament. *COM3800: Intro to Distributed Systems*. Sept. 1, 2019.

[18] Julian Barreiro-Gomez. "Distributed System Partitioning and DMPC". In: Springer International Publishing, Aug. 2, 2018, pp. 179–192. DOI: 10.1007/978-3-319-92204-1_9.

[19] Seyed Mehdi Meshkani and Bilal Farooq. *A Decentralized Shared CAV System Design and Application.* Apr. 17, 2021.

[20] Christine Schwaegerl, Geza Joos, and Nikos Hatziargyriou. "Active Distributed Systems and Distributed Energy Resources". In: Springer International Publishing, July 21, 2020, pp. 519–567. DOI: 10.1007/978-3-030-44484-6_15.

[21] R Chandrasekar, RK Suresh, and SG Ponnambalam. "Evaluating an obstacle avoidance strategy to ant colony optimization algorithm for classification in event logs". In: *2006 International Conference on Advanced Computing and Communications*. IEEE. 2006, pp. 628–629.

[22] R Sukharev, Oleg Lukyanchikov, Evgeny Nikulchev, Dmitry Biryukov, and Igor Ryadchikov. "Methods and tools for profiling and control of distributed systems". In: *IOP Conference Series: Materials Science and Engineering* 312.1 (Mar. 1, 2018), pp. 012024–. DOI: 10.1088/1757-899x/312/1/012024.

[23] Imran Siddique. *LibGuides: Distributed Systems: Home*. July 19, 2018.

[24] Jinrong Yang, Zimeng Wang, Rong Chen, and Haibo Chen. "A System-level Abstraction and Service for Flourishing AI-powered Applications". In: *Proceedings of the 16th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, Oct. 11, 2025, pp. 106–114. DOI: 10.1145/3725783.3764406.

[25] Cristian Márquez Zurita and Flávio Assis. "Hybrid Control Strategies for Greenhouse Climate Regulation: PID, Fuzzy, and Neuro-Fuzzy Comparative Implementation in Temperate-Dry Crop Systems". In: *2025 XV Symposium on Computing Systems Engineering (SBESC)*. IEEE, Nov. 24, 2025, pp. 1–6. DOI: 10.1109/sbesc68008.2025.11288904.

[26] George Coulouris and et. all. *Distributed Systems: Concepts and Design, 5/E*. Feb. 8, 2021.

[27] Chenlu Zhu, Xiaoxuan Fan, Xianjun Deng, Shenghao Liu, Duan Yin, Hanjun Gao, and Laurence T. Yang. "Area Coverage Reliability Evaluation for Collaborative Intelligence and Meta-Computing of Decentralized Industrial Internet of Things". In: *IEEE Internet of Things Journal* 12.10 (May 15, 2025), pp. 13734–13745. DOI: 10.1109/jiot.2025.3530941.

[28] Aleksey Charapko, Ailidani Ailijiang, Murat Demirbas, and Sandeep S. Kulkarni. "Retroscope: Retrospective Monitoring of Distributed Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 30.11 (Nov. 1, 2019), pp. 2582–2594. DOI: 10.1109/tpds.2019.2911944.

[29] *On Convex Embedding and Control Design for Nonlinear Homogeneous Systems*. June 29, 2021.

[30] Raghavendra Prasad M. "Dynamic Load Balancing in Distributed Systems". In: *International Journal for Research in Applied Science and Engineering Technology* 13.9 (Sept. 30, 2025), pp. 694–701. DOI: 10.22214/ijraset.2025.74018.

[31] Raju Kumar Mishra and Sundar Rajan Raman. "Introduction to PySpark SQL". In: Apress, Mar. 19, 2019, pp. 1–22. DOI: 10.1007/978-1-4842-4335-0_1.

[32] Jure Slak and Gregor Kosec. "FAST GENERATION OF VARIABLE DENSITY NODE DISTRIBUTIONS FOR MESH-FREE METHODS". In: *WIT transactions on engineering sciences* 122 (Sept. 11, 2018), pp. 163–173.

[33] Bishnu Kant Shukla, Bhupender Parashar, Vikas Kumar Singla, and Shivam Verma. *Autonomy and Adaptive Architectures in Distributed Systems*. Nov. 7, 2025. DOI: 10.1002/9781394288038.ch9.

[34] Markus Dahlmanns, Felix Heidenreich, Johannes Lohmöller, Jan Pennekamp, Klaus Wehrle, and Martin Henze. "Unconsidered Installations: Discovering IoT Deployments in the IPv6 Internet". In: *NOMS 2024-2024 IEEE Network Operations and Management Symposium*. IEEE, May 6, 2024, pp. 1–8. DOI: 10.1109/noms59830.2024.10574963.

[35] Chandra Yogatama and Randi Dwi Wibisono. "Investigating Quantum Approximation Techniques for Tackling NP-Hard Problems in Distributed Systems". In: *ALCOM: Journal of Algorithm and Computing* 1.1 (Feb. 10, 2025), pp. 33–44. DOI: 10.63846/emqcw506.

[36] Pascal Wichmann, Alexander Groddeck, and Hannes Federrath. "FileUploadChecker: Detecting and Sanitizing Malicious File Uploads in Web Applications at the Request Level". In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. ACM, Aug. 23, 2022, pp. 1–10. DOI: 10.1145/3538969.3538999.

[37] R Chandrasekar and T Srinivasan. "An improved probabilistic ant based clustering for distributed databases". In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI*. 2007, pp. 2701–2706.

[38] Yogesh Simmhan. "Pedagogic Practices for Teaching Distributed Systems Courses". In: *Proceedings of the 15th Annual ACM India Compute Conference*. ACM, Nov. 9, 2022, pp. 6–6. DOI: 10.1145/3561833.3568497.

[39] Filippo Poltronieri, Cesare Stefanelli, and Mauro Tortonesi. "Value-of-Information Middleware Solutions for Fog and Edge Computing". In: *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*. IEEE, Apr. 25, 2022, pp. 1–6. DOI: 10.1109/noms54207.2022.9789944.

[40] Emma Ahrens, Marius Bozga, Radu Iosif, and Joost-Pieter Katoen. *Reasoning about Reconfigurations of Distributed Systems*. Jan. 1, 2021. DOI: 10.48550/arxiv.2107.05253.

[41] Kurt Klaus Horvath, Dragi Kimovski, Bernd Spiess, Oliver Hohlfeld, and Radu Prodan. "SEAL-CC: Scalable Latency Evaluation Methodology for Internet-of-Things Services". In: *Proceedings of the 14th International Conference on the Internet of Things*. ACM, Nov. 19, 2024, pp. 117–126. DOI: 10.1145/3703790.3703804.

[42] Wan Nor Shuhadah Wan Nik. "A Survey on Data Replication and Resource Management in Distributed Systems". In: *International Journal of Advanced Trends in Computer Science and Engineering* (Oct. 15, 2019), pp. 2638–2646. DOI: 10.30534/ijatcse/2019/117852019.

[43] Olena Pavliuk, Mykola Medykovskyy, Rafal Cupek, and Myroslav Mishchuk. "Modern Methods of Data Preprocessing to Increase the Accuracy of AGV Battery Discharge Forecast". In: *2024 IEEE 19th International Conference on Computer Science and Information Technologies (CSIT)*. IEEE, Oct. 16, 2024, pp. 1–4. DOI: 10.1109/csit65290.2024.10982564.

[44] David Basin, Matthieu Gras, Srdan Krstic, and Joshua Schneider. "RV - Scalable Online Monitoring of Distributed Systems". In: Germany: Springer International Publishing, Oct. 2, 2020, pp. 197–220. DOI: 10.1007/978-3-030-60508-7_11.

[45] Oleg Savenko, Anatoliy Sachenko, Sergii Lysenko, George Markowsky, and Nadiia Vasylkiv. "BOTNET DETECTION APPROACH BASED ON THE DISTRIBUTED SYSTEMS". In: *International Journal of Computing* 19.2 (June 14, 2020), pp. 190–198. DOI: 10.31891/1727-6209/2020/19/2-190-198.

[46] Edoardo Di Caro and Mauro Tortonesi. "Machine Learning for Performance Optimization in Resource-Constrained Environments". In: *2025 21st International Conference on Network and Service Management (CNSM)*. IEEE, Oct. 27, 2025, pp. 1–4. DOI: 10.23919/cnsm67658.2025.11297462.

[47] Nikolay Raychev. *System architecture for maintenance of complex distributed systems*. July 16, 2020. DOI: 10.37686/ser.v1i2.60.

[48] Justus Breyer, Sparsh Jauhari, René Glebke, Muhammad Hamad Alizai, Markus Stroot, and Klaus Wehrle. "Investigating Domain Bias in NILM". In: *Proceedings of the 11th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*. ACM, Oct. 29, 2024, pp. 333–336. DOI: 10.1145/3671127.3699532.

[49] Xiaoniu Song, Rong Chen, Haitao Song, Yiwen Zhang, and Haibo Chen. "Unified and Near-optimal Multi-GPU Cache for Embedding-based Deep Learning". In: *ACM Transactions on Computer Systems* 44.1 (Nov. 7, 2025), pp. 1–32. DOI: 10.1145/3767725.

[50] Peter Simon Sapaty. "Managing Distributed Systems with Spatial Grasp Patterns". In: CRC Press, Feb. 27, 2024, pp. 127–145. DOI: 10.1201/9781003425267-7.

[51] Ion-Alexandru Secara. "Challenges and Considerations in Developing and Architecting Large-scale Distributed Systems". In: B P International (a part of SCIENCEDOMAIN International), Apr. 8, 2023, pp. 1–15. DOI: 10.9734/bpi/rhmcs/v8/5760a.

[52] Ognjen Scekic, Mirela Riveni, Hong-Linh Truong, and Schahram Dustdar. "Social Interaction Analysis for Team Collaboration". In: Springer New York, June 12, 2018, pp. 2607–2621. DOI: 10.1007/978-1-4939-7131-2_257.

[53] Vivek Vijaykumar, R Chandrasekar, and T Srinivasan. "An obstacle avoidance strategy to ant colony optimization algorithm for classification in event logs". In: *2006 IEEE Conference on Cybernetics and Intelligent Systems*. IEEE. 2006, pp. 1–6.

[54] Akash Samanta, Chandan Chetri, Dominic Karnehm, Antje Neve, and Sheldon Williamson. "Temporal Sensitivity Analysis of Internal Temperature Informed Charging Algorithms and Rapid Thermal Management System for E-mobility". In: *2024 IEEE Energy Conversion Congress and Exposition (ECCE)*. IEEE, Oct. 20, 2024, pp. 2302–2304. DOI: 10.1109/ecce55643.2024.10861466.

[55] Ahmad Shukri Mohd Noor, Nur Farhah Mat Zian, and Fatin Nurhanani M. Shaiful Bahri. "SURVEY ON REPLICATION TECHNIQUES FOR DISTRIBUTED SYSTEM". In: *International Journal of Electrical and Computer Engineering (IJECE)* 9.2 (Apr. 1, 2019), pp. 1298–1303. DOI: 10.11591/ijece.v9i2.pp1298-1303.

[56] Marek R. Ogiela and Urszula Ogiela. "AI for Security of Distributed Systems". In: *WSEAS TRANSACTIONS ON COMPUTER RESEARCH* 12 (Oct. 21, 2024), pp. 450–454. DOI: 10.37394/232018.2024.12.44.

[57] Alin-Mihai Căilean, Cătălin Beguni, and Mihai Dimian. "A Novel Approach for Improved Precision Ranging Based on Vehicle-to-Vehicle Visible Light Communications: Concept and Experimental Demonstration". In: *2025 International Conference on Applied Electronics (AE)*. IEEE, Sept. 8, 2025, pp. 1–4. DOI: 10.1109/ae66163.2025.11197801.

[58] Vivek Kale. "Distributed Systems". In: CRC Press, July 8, 2019, pp. 71–94. DOI: 10.1201/9781351029148-5.

[59] Eric Wagner, Frederik Basels, Jan Bauer, Till Zimmermann, Klaus Wehrle, and Martin Henze. "Caiba: Multicast Source Authentication for CAN Through Reactive Bit Flipping". In: *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. IEEE, June 30, 2025, pp. 710–719. DOI: 10.1109/eurosp63326.2025.00045.

[60] Viktor Matkovic, Malte Josten, and Torben Weis. "Optical Token Transmission for Secure IoT Device Discovery and Control in Ubiquitous Environments". In: *Companion of the 2025 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, Oct. 12, 2025, pp. 1530–1533. DOI: 10.1145/3714394.3756299.

[61] Bruna Henning Pereira and Cesar Albenes Zeferino. "An Embedded System for Predicting Epileptic Seizures Using Machine Learning". In: *2025 23rd IEEE Interregional NEWCAS Conference (NEWCAS)*. IEEE, June 22, 2025, pp. 276–280. DOI: 10.1109/newcas64648.2025.11107127.

[62] T Srinivasan, R Chandrasekar, Vivek Vijaykumar, V Mahadevan, A Meyyappan, and A Manikandan. "Localized Tree Change Multicast Protocol for Mobile Ad Hoc Networks". In: *2006 International Conference on Wireless and Mobile Communications (ICWMC'06)*. IEEE. 2006, pp. 44–44.

[63] Chen Gao, Yujiang Zhong, and Xiao He. "Safety Control of Distributed Systems". In: Elsevier, Jan. 1, 2024, pp. 153–159. DOI: 10.1016/b978-0-443-14081-5.00045-3.

[64] Sachin Shetty, Charles A. Kamhoua, and Laurent Njilla. *Blockchain for Distributed Systems Security*. Feb. 10, 2020.

[65] Rastko Gajanin, Cynthia Marcelino, and Stefan Nastić. "Performance Isolation for Serverless Functions". In: *IEEE Transactions on Services Computing* (Jan. 1, 2025), pp. 1–18.

[66] Roman Matzutt, Vincent Ahlrichs, Jan Pennekamp, Roman Karwacik, and Klaus Wehrle. "A Moderation Framework for the Swift and Transparent Removal of Illicit Blockchain Content". In: *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, May 2, 2022, pp. 1–9. DOI: 10.1109/icbc54727.2022.9805508.

[67] Asier Estevan. "An Approach to Distributed Systems from Orderings and Representability". In: *Bulletin of the Iranian Mathematical Society* 50.3 (Apr. 8, 2024). DOI: 10.1007/s41980-024-00865-0.

[68] Pascal Weisenburger and Guido Salvaneschi. "DEBS - Developing Distributed Systems with Multitier Programming". In: *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*. ACM, June 24, 2019, pp. 203–204. DOI: 10.1145/3328905.3332465.

[69] SHARIFAH HAFIZAH SY AHMAD UBAIDILLAH, NORAZIAH AHMAD, and Noor Azida Sahabudin. "A survey on potential reactive fault tolerance approach for distributed systems in big data". In: *Third International Conference on Computer Vision and Information Technology (CVIT 2022)*. SPIE, Feb. 24, 2023, pp. 21–21. DOI: 10.1117/12.2670017.

[70] Rahul Malik, Chandrasekar Ramachandran, Indranil Gupta, and Klara Nahrstedt. "Samera: a scalable and memory-efficient feature extraction algorithm for short 3D video segments." In: *IMMERSCOM*. 2009, p. 18.

[71] Michael F. Rehme, Stefan Zimmer, and Dirk Pflüger. "Hierarchical Extended B-splines for Approximations on Sparse Grids". In: Germany: Springer International Publishing, Oct. 22, 2021, pp. 187–203. DOI: 10.1007/978-3-030-81362-8_8.

[72] Egon Börger and Alexander Raschke. "Modeling Distributed Systems". In: Springer Berlin Heidelberg, Apr. 1, 2018, pp. 207–239. DOI: 10.1007/978-3-662-56641-1_6.

[73] Kai Cui, Shenghao Liu, Wei Feng, Xianjun Deng, Liangbin Gao, Minmin Cheng, Hongwei Lu, and Laurence T. Yang. "Correlation-Aware Cross-Modal Attention Network for Fashion Compatibility Modeling in UGC Systems". In: *ACM Transactions on Multi-*

media Computing, Communications, and Applications 21.11 (Nov. 10, 2025), pp. 1–24. DOI: 10.1145/3698772.