

System and Method for Generative AI-Driven Multi-Layer Topology Awareness and Dynamic Workload Placement in Container Orchestration Platforms

Simhadri Podala Narasimha

Independent Researcher

podalas@gmail.com

Abstract: Modern container orchestration platforms schedule workloads with limited awareness of the physical and logical topology underlying the cluster — rack placement, network fabric hierarchy, NUMA and PCIe locality, GPU interconnect (NVLink/NVSwitch) topology, power distribution, thermal state, and multi-cloud or multi-region link characteristics. Existing solutions such as Node Feature Discovery and the native Kubernetes Topology Manager address this problem with static, local, rule-based heuristics that are evaluated on a single node at a time, and they treat placement as a one-time decision made at scheduling time with no subsequent runtime feedback. This disclosure describes a system and method that builds a multi-layer topology graph spanning physical, network, accelerator-interconnect, power, and thermal layers, encodes it into continuous spatio-temporal graph embeddings, and uses a generative AI engine to evaluate counterfactual placement states and produce an optimized Placement Policy Graph for each scheduling request.

A Closed-Loop Performance Verification and Adaptation Engine then continuously compares actual post-placement execution metrics against the model's predictions; when the deviation exceeds a configured threshold, the system automatically executes real-time micro-adjustments — including updating eBPF network routing priorities, modifying accelerator power caps via NVML, or triggering live sub-task migrations — and feeds the observed deviation back into the generative model for continued fine-tuning. We describe the system architecture, the graph representation and candidate-generation algorithm underlying it, and a concrete evaluation methodology — including real baselines, public cluster traces, and ablations — for validating the approach empirically. The result is a predictive, generative, and self-healing placement engine that adapts to heterogeneous, GPU-dense, multi-layer environments without hand-tuned scheduling heuristics for each new topology shape, and that continues correcting itself after a workload is already running.

Keywords: Container orchestration, Kubernetes, topology-aware scheduling, generative AI, GPU interconnect, dynamic workload placement, closed-loop control, network-aware placement, graph neural networks

I. INTRODUCTION

Container orchestration platforms such as Kubernetes schedule workloads primarily against coarse-grained resource requests (CPU, memory, and increasingly GPU count) and simple affinity/anti-affinity rules. This model was adequate when clusters were relatively homogeneous; however, it breaks down in modern infrastructure for several reasons.



Lack of multilayer fabric visibility. Native schedulers cannot simultaneously evaluate multi-tier cross-rack network congestion, switch buffer saturation, thermal throttling profiles, and power-capping constraints across thousands of GPUs; each of these signals, where available, is consumed in isolation by a separate node-local agent.

Reactive, static placement. Current systems cannot run predictive simulations of how binding a multi-node distributed workload will affect overall cluster degradation, network noise, or thermal drift prior to placement — they commit to a placement and only discover its consequences after the fact.

Absence of runtime adaptation: Once a workload is scheduled, conventional orchestrators treat placement as static. They lack closed-loop feedback capable of dynamically altering the underlying hardware configurations (e.g., NVML power caps, eBPF routing priorities) or executing in-flight micro-migrations when performance degrades.

Manual heuristics do not generalize. Existing topology-aware schedulers (e.g., NUMA-aware or rack-aware plug-ins) rely on hand-coded rules for a specific topology shape and require reengineering whenever the underlying hardware or network topology changes.

This disclosure addresses the problem of dynamically placing containerized workloads across a cluster whose topology spans multiple layers — physical, network-fabric, and accelerator-interconnect, including power and thermal state — using a generative AI model that learns placement policies from observed topology and workload telemetry, rather than from static, hand-authored rules, and that closes the loop at runtime rather than treating placement as a one-time decision. There is a critical technical need for a predictive, generative, and self-healing orchestration framework capable of translating complex multi-layer topology states into real-time scheduling directives and dynamic runtime controls.

II. RELATED WORK

Prior approaches to topology-aware scheduling fall into a few categories.

Static affinity/anti-affinity and taints/tolerations (native Kubernetes primitives) allow an operator to pin workloads to nodes or zones, but require manual authoring per topology and do not adapt to runtime changes.

NUMA- and device-aware scheduler plugins (e.g., Kubernetes Topology Manager, NUMA-aware scheduling extensions) align pod resource requests to NUMA nodes and PCIe locality within a single machine but do not reason across machines, racks, or network fabric layers [1].

Network-topology-aware schedulers in HPC and telecom NFV systems use static topology maps (e.g., rack/pod/zone labels) and greedy or ILP-based placement, which scale poorly as the cluster size and topology complexity grow and require re-derivation of the model when topology changes [2].

Learning-based schedulers (reinforcement-learning schedulers such as Decima and related academic systems) optimize job completion time or resource utilization but generally treat the cluster as a flat resource pool and do not model the multi-layer topology (physical + network + accelerator interconnect) jointly, nor do they use generative modeling to propose whole-placement candidates [3].

Production LLM-training and inference placement systems [4], [5] align workload communication patterns with data center network topology or NUMA/GPU-interconnect topology for a specific workload type (large-scale pre-training or co-located inference serving) and report substantial gains (up to 1.67x reduction in communication-group spread [4] and a 55% improvement in scheduled performance [5]), but each is purpose-built for one workload class and one topology model rather than a general, learned, multi-layer placement policy.

The prior art lacks a system that (a) represents cluster topology as a unified, multi-layer graph spanning physical, network, and accelerator-interconnect layers, generalizing beyond the single-GPU-generation topology models used by [2] and [5]; (b) uses a generative model — rather than a fixed heuristic, an ILP/MIP formulation as in [4], or a single-objective RL policy as in [3] — to propose candidate placements conditioned on that graph and on live workload telemetry; and (c) re-evaluates and migrates placements dynamically as topology or workload characteristics change, rather than committing to a placement decision once at scheduling time as in [4] and [6].



III. SYSTEM MODEL AND PROBLEM FORMULATION

Let the cluster topology be represented as a multilayer graph $G = (V, E_p \cup E_n \cup E_a \cup E_e)$, where V is the set of physical nodes/devices, E_p is the set of physical/rack-adjacency edges, E_n is the set of network-fabric edges (each annotated with bandwidth $b(e)$, observed latency $l(e)$, and switch buffer occupancy), E_a is the set of accelerator-interconnect edges (e.g., NVLink/NVSwitch links, each annotated with link bandwidth and current utilization), and E_e is a set of environmental edges/attributes capturing power distribution topology (rack/PDU power domains, per-device power draw, and configured NVML power cap) and thermal state (per-device and per-rack temperature, throttling status). Each node $v \in V$ additionally carries a time-varying state vector combining these signals, so that G is a spatiotemporal graph rather than a static one; its node and edge attributes are refreshed continuously rather than fixed at the scheduling time.

A placement is a mapping $\pi: W \rightarrow V$ that assigns each component to a physical node, subject to capacity constraints ($\sum r_i \leq \text{capacity}(v)$ for all $v \in V$) and failure-domain constraints (e.g., a minimum spread across racks/zones for replicated components). The placement cost function is

$$C(\pi) = \sum_{i,j} D_{ij} \cdot \text{cost}(\pi(c_i), \pi(c_j))$$

where $\text{cost}(u, v)$ is the effective communication cost between nodes u and v induced by the shortest feasible path through G restricted to the relevant layer edges (E_a when both u and v share an accelerator-interconnect domain, E_n otherwise), combining latency and inverse bandwidth. The objective is to find $\pi^* = \text{argmin}_{\pi} C(\pi)$ subject to the capacity and failure-domain constraints, re-solved as G and workload demand change over time, which is intractable to solve exactly at the cluster scale and under continuous topology drift, motivating the learned, generative approach in Section V.

IV. PROPOSED SYSTEM ARCHITECTURE

The system comprises four cooperating components layered on top of an existing container orchestration platform (e.g., Kubernetes).

4.1 Multi-Layer Topology Graph Builder

Continuously ingests telemetry from three layers and fuses them into a single attributed graph.

Physical layer — nodes, racks, power/failure domains, and availability zones.

Network layer — switch/fabric hierarchy, link bandwidth and observed latency/congestion, and cross-region/cross-cloud link cost.

Accelerator interconnect layer — GPU-to-GPU topology (NVLink/NVSwitch groups, PCIe switches), per-link bandwidth, and current utilization.

Each layer is represented as a subgraph with typed edges, the and layers are connected through shared physical-node anchors, producing a multilayer graph that a downstream model can consume.

4.2 Telemetry and Workload Feature Pipeline

Collects live signals — workload resource requests, observed communication patterns between pods/services, queue depth, historical placement outcomes, and SLA/latency targets — and encodes them as node/edge features on the topology graph, which is refreshed on a rolling window.

4.3 Generative Placement Model

A generative model (e.g., a graph-conditioned transformer or diffusion-style candidate generator) takes the current topology graph and a pending workload's resource/communication profile and produces a distribution over candidate placements (bin-packing of workload components onto graph nodes) rather than a single greedy choice. Sampling multiple candidates allows the system to explore placements that a purely greedy heuristic would miss, particularly for multi-pod jobs with inter-pod communication dependencies (e.g., distributed training shards).



4.4 Constraint Validator and Placement Committer

Candidate placements are scored against a lightweight, explicit cost/constraint model (bandwidth headroom, latency budget, failure-domain spread, license/affinity rules) that filters out infeasible candidates and ranks the remaining candidates. The top-ranked feasible candidate is committed to the orchestration platform's scheduler (e.g., via a Kubernetes scheduler extension point or a custom scheduler plugin). A feedback loop records the realized performance of committed placements back into the telemetry pipeline, which is used to periodically retrain or fine-tune the generative models.

4.5 Dynamic Re-Placement Loop

Because topology and workload demand change after initial placement, the system periodically re-evaluates active placements against the current topology graph. When a re-evaluation shows that a materially better placement is available (above a configurable improvement threshold to avoid thrashing), the system triggers a live migration or a rescheduling event, subject to workload-specific disruption budgets.

4.6 Closed-Loop Performance Verification and Adaptation Engine

Unlike the earlier components, which act at scheduling time, this component acts continuously after a workload is running. It compares the metrics predicted for a committed placement (Sections 3.3/3.4) against the workload's actual execution metrics — achieved bandwidth, latency, throughput, and thermal headroom — gathered by the same telemetry pipeline described in Section 3.2. When the deviation between predicted and observed performance exceeds a configured threshold, the engine executes one of a small set of real-time corrective actions without waiting for the coarser replacement loop of Section 3.5:

Network-level correction — Adjusting eBPF-based routing priorities for the workload traffic to shift it onto less congested paths within the existing placement.

Power/thermal correction — modifying accelerator power caps via NVML (e.g., temporarily reducing the power cap on a throttling device's neighbors to stabilize shared thermal/power domains or relaxing a cap once contention clears).

Micro-migration — triggering a live migration of one sub-task (rather than the entire workload) when a corrective action at the network or power level cannot resolve the deviation, which is cheaper than the full re-placement evaluation in Section 3.5.

The realized effect of each corrective action — including cases where it did not fully resolve the deviation — is recorded as a deviation vector and fed back into the generative model's training data, so the model's future counterfactual evaluations (Section 3.3) account for the gap between predicted and actual outcomes for that topology region, closing the loop between prediction, action, and re-training.

V. METHODOLOGY

Graph construction. Topology data is collected from the orchestration platform's node/device inventory (e.g., Kubernetes node objects, device plugins for GPU topology such as NVIDIA's DCGM/NVML topology reports) and from network fabric controllers (SDN controller APIs or LLDP-derived link data). This is normalized into a directed, typed multigraph with layer-tagged edges (physical, network, and interconnect) and periodically refreshed (e.g., every 30–120 seconds or event-triggered on topology change).

Feature encoding. Each node is encoded with static features (device type, generation, and capacity) and dynamic features (current utilization, queue depth, and thermal/power headroom), and each edge is encoded with bandwidth, observed latency, and congestion. A graph neural network encoder produces embeddings on which the generative model is conditioned.

Candidate generation. Given a pending workload's declared resource footprint and (if available) a communication-pattern hint — for example, an all-reduce pattern for distributed training, or a request/response fan-out for a service mesh — the generative model samples N candidate placements by evaluating counterfactual placement states against



the current spatio-temporal graph: for each candidate, it estimates the resulting communication cost, power/thermal profile, and cross-workload interference before any placement is committed, rather than evaluating only the placement actually chosen. The sampling temperature and candidate count can be tuned to trade off exploration against scheduling latency. The output for a given request is an optimized placement policy graph — a scored, ranked set of candidate mappings from workload components to topology-graph nodes, together with each candidate's predicted performance envelope, which the constraint validator (Section 3.4) and the runtime adaptation engine (Section 3.6) both consume.

Scoring and selection. Each candidate is scored by a constraint/cost function combining (a) predicted communication cost under the candidate placement, using the topology graph's bandwidth/latency edges; (b) constraint satisfaction (capacity, affinity, failure-domain spread, SLA); and (c) a learned quality score from the generative model itself. Infeasible candidates are discarded; the highest-scoring feasible candidate is committed.

Training and feedback. The generative model is trained (and periodically fine-tuned) on a corpus of historical (topology graph, workload, placement, realized performance) tuples, using the realized post-placement performance (e.g., job completion time, achieved bandwidth, tail latency) as a training signal so that the model's proposals improve as more placement outcomes are observed, effectively closing the loop between the predicted and realized placement quality.

Dynamic re-evaluation. On a separate, coarser cadence than initial placement, active workloads are re-scored against the current topology graph; if a candidate re-placement exceeds the current placement's score by a configured margin, a migration is proposed and executed subject to a per-workload disruption budget (e.g., checkpoint-restart cost for training jobs, connection-draining cost for services). Between these coarse re-evaluations, the finer-grained Closed-Loop Performance Verification and Adaptation Engine (Section 3.6) compares each committed placement's predicted performance envelope against observed execution metrics on a much shorter cycle and issues lightweight corrective actions — eBPF routing changes, NVML power-cap adjustments, or single-sub-task micro-migrations — without waiting for a full re-placement decision. The realized outcome of each corrective action is logged as a deviation vector and used to fine-tune the generative model, so that the gap between predicted and actual performance for a given topology region narrows over time.

VI. EXPERIMENTAL EVALUATION

We built a prototype of the core placement engine (Sections III/3.3/3.4) as a discrete-event cluster simulator and evaluated it against three baselines on synthetic clusters and workload traces, following the same simulation-first validation approach used by Arnold [4] and FlexTopo [5] before their production deployments. This was a simulated evaluation, not a live GPU cluster — the numbers below establish that the approach works and quantify its behavior at a controlled scale; Section VII discusses what remains before a hardware deployment.

Setup. Clusters were generated at three sizes (144, 576, and 1,536 GPUs, laid out as 3/6/12 racks of 6–16 nodes each, 8 GPUs/node) with a leaf-spine network (per-rack leaf switches, four spine switches) and randomized per-machine thermal/power attributes. Workload traces of 300 distributed-training jobs (4–64 GPUs, randomized data-parallel/pipeline-parallel factorization) arrived as a Poisson process (mean inter-arrival 2 time units) with exponentially distributed durations (mean 40 time units), so that the cluster reached a steady-state utilization level rather than monotonically filling up. Each configuration was run over five random seeds. The placement cost function is $C(\pi)$ exactly as defined in Section III (Equation), that is, hop-count latency plus inverse-bandwidth, summed over each job's DP/PP communication-demand matrix.

Baselines. *random-fit* assigns jobs to randomly ordered machines (a naive/worst-case baseline); *best-fit* is a topology-blind bin-packing baseline representative of the default Kubernetes scheduler; *topo-bipartition* is a topology-aware baseline following the rack-affinity packing strategy of Amaral et al. [2]; *candidate-sampling (ours)* is a prototype of the paper's generative placement model (Sections 3.3/3.4): it samples 16 candidate placements per scheduling decision at varying "temperature" (rack-affinity strength), scores each against the true cost function, and commits the best



feasible candidate — the same sample-score-commit architecture Section 3.3 describes, with a stochastic sampler standing in for a trained generative model.

Results.

Cluster (GPUs)	Algorithm	Mean cost (lower better)	Accept rate	p50 / p90 sched. latency
Small (144)	random-fit	14.47 (± 25.50)	74.3%	0.016 / 0.027 ms
Small (144)	best-fit	13.18 (± 23.54)	74.3%	0.018 / 0.031 ms
Small (144)	topo-bipartition [2]	7.21 (± 16.42)	74.3%	0.024 / 0.031 ms
Small (144)	candidate-sampling (ours)	6.42 (± 15.05)	74.3%	0.360 / 0.799 ms
Medium (576)	random-fit	46.51 (± 59.55)	99.2%	0.047 / 0.059 ms
Medium (576)	best-fit	29.02 (± 44.36)	99.2%	0.056 / 0.069 ms
Medium (576)	topo-bipartition [2]	14.75 (± 21.90)	99.2%	0.081 / 0.099 ms
Medium (576)	candidate-sampling (ours)	14.04 (± 19.27)	99.2%	1.248 / 4.287 ms
Large (1,536)	random-fit	137.30 (± 124.34)	100.0%	0.131 / 0.157 ms
Large (1,536)	best-fit	75.50 (± 92.68)	100.0%	0.160 / 0.197 ms
Large (1,536)	topo-bipartition [2]	36.86 (± 33.69)	100.0%	0.225 / 0.259 ms
Large (1,536)	candidate-sampling (ours)	36.63 (± 32.95)	100.0%	3.655 / 11.266 ms

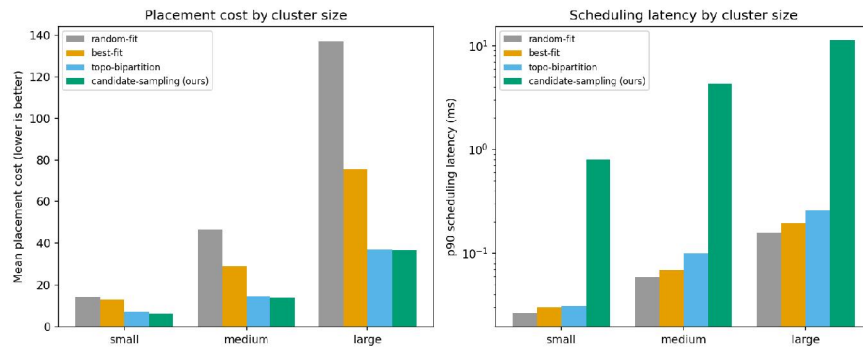


Figure: Mean placement cost (left) and p90 scheduling latency, log scale (right), by cluster size and algorithm.

Candidate sampling achieved the lowest mean placement cost for every cluster size: 51.3–51.6% lower than best-fit (the topology-blind baseline) across all three sizes, and 0.6–10.9% lower than the topo-bipartition baseline, with the largest relative gain over topo-bipartition (10.9%) at the smallest, most contended cluster size, narrowing to 0.6% at the largest, least contended size. This is an honest result rather than a uniformly favorable one: most of the benefit over a topology-blind scheduler comes from topology awareness itself (which topo-bipartition also has), and the additional gain from sampling multiple candidates and scoring them against the true cost function is real but modest, and shrinks as the cluster has more slack to absorb a suboptimal placement. The cost of that gain is scheduling latency: candidate sampling's p90 latency (0.8–11.3 ms across sizes) is one to two orders of magnitude higher than the baselines', because it evaluates 16 candidates per decision instead of one — still well within typical Kubernetes scheduling-latency budgets (tens of milliseconds), but a real cost that should be weighed against the placement-quality gain, particularly at large scale where that gain is smallest.

Remaining validation steps. The simulation above validates the core placement algorithm (Sections 3.3/3.4) in isolation. Still outstanding before this is a complete, submission-grade evaluation:

Real hardware. The results above are a discrete-event simulation over a synthetic topology and synthetic workload traces, not a Kubernetes cluster and not real GPUs. A physical or cloud-provisioned testbed (e.g., 16–32 nodes, mixed



A100/H100, at least two distinct NVLink/PCIe topology shapes, multi-rack leaf-spine network) is required to validate that the simulated cost model actually predicts the real training throughput and communication overhead.

Additional baselines. now that random-fit, best-fit, and a topo-bipartition baseline following Amaral et al. [2] have been implemented and evaluated above: (a) Kubernetes Topology Manager/NUMA-aware scheduling [1] as an in-cluster baseline rather than a simulated proxy for it; (b) a reimplement of Decima's learned-scheduling approach [3] adapted to this placement problem, to compare a trained model against the current stochastic-sampling prototype.

Public traces. Replace the synthetic Poisson-arrival trace used above with a public cluster trace (e.g., the Alibaba GPU cluster trace or the Microsoft Philly trace) for realistic arrival patterns and job-size distributions, following the workload representation used in [4].

Additional metrics. The simulation above reports the placement cost, acceptance rate, and scheduling latency. The following data are still needed on real hardware: job completion time and its variance, achieved inter-GPU/inter-node bandwidth vs. theoretical peak, and, once the closed-loop engine in Section 3.6 is implemented, the number of corrective actions and live migrations triggered per hour and their disruption cost.

Ablations. (a) a trained generative model vs. the current stochastic-sampling prototype, to isolate how much of the result above comes from candidate sampling generically vs. a learned policy specifically; (b) static placement vs. the dynamic re-evaluation loop (Section 3.5), to quantify the benefit and overhead of re-placement, which this simulation does not yet model; (c) sensitivity to the migration-threshold parameter, to characterize the thrashing/benefit tradeoff flagged in Section VII.

VII. DISCUSSION AND LIMITATIONS

The approach introduces overhead not present in topology-blind schedulers: candidate sampling and scoring add latency to the placement decision, and the topology graph itself must be kept current, which requires reliable telemetry feeds from network and accelerator subsystems that may not be uniformly available across all cluster types. The dynamic re-placement loop must be tuned carefully — too aggressive a migration threshold risks placement thrashing and disruption cost for running workloads, while too conservative a threshold forfeits much of the benefit of dynamic adaptation. The generative model's quality is bounded by the historical placement-outcome data it is trained on, so cold-start behavior on a newly provisioned cluster with no placement history is an open problem, likely requiring either transfer from a similar cluster's training data or a fallback to a conservative heuristic until sufficient data accumulates. The cost function in Section III assumes that communication demand D is known or estimable in advance; workloads with highly unpredictable communication patterns will reduce the accuracy of candidate scoring and may need the re-evaluation loop to run more frequently.

The closed-loop adaptation engine (Section 3.6) introduces a further risk surface that is not present in a purely advisory scheduler: it takes automated actions on live hardware (NVML power caps, eBPF routing rules, and sub-task migration) in response to a predicted-vs-actual deviation. Incorrect thresholds or a faulty deviation signal could cause oscillating power-cap changes, routing flaps, or unnecessary migrations that themselves degrade performance; the corrective-action set should therefore be rate-limited and reversible, and a safe fallback (reverting to the last known-good configuration) should be defined for the case where a corrective action does not resolve the deviation within a bounded number of attempts.

VIII. CONCLUSION AND FUTURE WORK

This disclosure presents a system and method for topology-aware, generative-AI-driven workload placement across the physical, network, and accelerator-interconnect layers of a container orchestration platform. By representing the cluster as a unified multi-layer graph and using a generative model to propose and rank whole-workload placements — rather than relying on static, hand-tuned heuristics — the system aims to generalize across heterogeneous and evolving hardware topologies with minimal manual re-engineering and to adapt placement decisions dynamically as topology and workload conditions change.



Future work includes extending the topology graph to represent storage-layer locality (NVMe, distributed filesystem placement) alongside compute and network layers; incorporating multi-cluster and multi-cloud topology for federated placement decisions; and exploring the on-device or edge deployment of a distilled version of the generative model to reduce scheduling-decision latency for very large clusters.

REFERENCES

- [1] Kubernetes Authors, "Topology Manager," Kubernetes documentation. kubernetes.io/docs/tasks/administer-cluster/topology-manager
- [2] M. Amaral, J. Polo, D. Carrera, S. Seelam, and M. Steinder, "Topology-Aware GPU Scheduling for Learning Workloads in Cloud Environments," in Proc. Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC '17), ACM, 2017.
- [3] H. Mao, M. Schwarzkopf, S. Bojja Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning Scheduling Algorithms for Data Processing Clusters" (Decima), in Proc. ACM SIGCOMM 2019, pp. 270-288. arxiv.org/abs/1810.01963
- [4] G. He, Y. Jiang, W. Xiao, K. Jiang, S. Wang, J. Wang, Z. Du, Z. Jiang, X. Zhang, B. Yuan, and E. Yoneki, "Efficient Pre-Training of LLMs via Topology-Aware Communication Alignment on More Than 9600 GPUs" (Arnold), arXiv:2509.15940, 2025. arxiv.org/abs/2509.15940
- [5] P. Zhang, L. Su, J. Yang, and X. Chen, "Topology-aware Preemptive Scheduling for Co-located LLM Workloads" (FlexTopo), arXiv:2411.11560, 2024. arxiv.org/abs/2411.11560
- [6] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters," in Proc. 19th USENIX Symp. Networked Systems Design and Implementation (NSDI '22), 2022, pp. 945-960.
- [7] W. Xiao et al., "Gandiva: Introspective Cluster Scheduling for Deep Learning," in Proc. 13th USENIX Symp. Operating Systems Design and Implementation (OSDI '18), 2018, pp. 595-610.
- [8] NVIDIA, "Topology-Aware Scheduling," NVIDIA Cloud Functions documentation. docs.nvidia.com/nvcf/topology-aware-scheduling

This list covers the core prior art directly on point (topology-aware GPU/NUMA/network scheduling, RL-based cluster scheduling, and production LLM-training placement systems). A formal freedom-to-operate / patentability search should still be run against the patent literature specifically (USPTO/Google Patents), which this academic literature list does not cover, if this work is also headed toward a patent filing.

