Symphony as Compute Ontology: Extending Insight into OpenShift and NVIDIA AI Factories

Technical Paper
April 6, 2026

Kevin D. Johnson, MBA, MAcc, MSGTD, MAT

DOI: 10.5281/zenodo.19890725

***Abstract*—OpenShift provides the enterprise standard for container compute, yet the container orchestration model offers no compute ontology: no typed, governed, semantically rich model of what workloads are doing, how well they are doing it or who governs what they do. NVIDIA's acquisition of Run:ai positions a GPU-aware container placer as the inference layer for its AI factory ecosystem but Run:ai only enhances placement decisions without modeling the operational state of the workloads it places. The present paper, the first of two parts, presents IBM Spectrum Symphony as the compute ontology extending insight into OpenShift and NVIDIA AI factory infrastructure. Symphony's compute ontology provides typed resource metrics with semantic direction through ELIM, hierarchical consumer governance with sub-second rebalancing, SOAM service lifecycle management with per-phase failure policies and cross-substrate routing spanning OpenShift clusters, bare-metal GPU hosts, cloud burst instances and heterogeneous accelerators under one workload management domain. A comparison with Kueue (v0.17) demonstrates that Kubernetes' strongest governance extension operates at alpha API maturity with zero capability in five of six ontological dimensions. A feature-complete analysis of Run:ai (v2.24) establishes that every Run:ai capability is replicable within Symphony's ELIM architecture and that Run:ai exclusively uses public GPU APIs with no proprietary hardware access. The architecture positions Symphony as the compute parallel to Palantir Foundry's data ontology, grounded in Songnian Zhou's 1987 doctoral research and four decades of production deployment. A companion paper (Part II) will present empirical validation across three compute substrates.**

I. Introduction

Kubernetes has become the enterprise standard for platform compute. Red Hat OpenShift, the enterprise Kubernetes distribution, is deployed across financial services, healthcare, government and telecommunications as the foundation for containerized application infrastructure. IBM's acquisition of Red Hat in 2019 made OpenShift central to IBM's hybrid cloud strategy [29]. The platform manages container lifecycle, networking, security, service discovery and CI/CD integration with a maturity no competing container orchestration system matches.

The AI factory is being built on the Kubernetes and OpenShift foundation. NVIDIA's reference architecture for enterprise AI infrastructure positions Kubernetes as a compute substrate, with Run:ai providing GPU-aware scheduling within the Kubernetes framework [1], [2]. Enterprises deploying GPU infrastructure for training and inference are deploying it on OpenShift and Kubernetes because that is where their platform engineering, operational expertise and security certification reside. The hardware decision (NVIDIA GPUs) is settled. The platform decision (Kubernetes/OpenShift) is settled. The workload management decision is not.

The workload management gap exists because Kubernetes is a container placement engine, not a compute ontology. Kubernetes answers four questions about the compute domain, namely what containers exist, what resources they requested, where they are running and whether they are alive. The four answers are placement facts. Placement facts describe the location and the request of each container. Placement facts do not describe what something is doing, how well it is doing it, what it should be doing or who governs what it does. The gap is categorical, not incremental. No amount of Kubernetes feature development closes it because the gap arises from Kubernetes' architectural position as a container placement engine rather than as a typed, governed model of the compute domain.

NVIDIA's response to the gap is Run:ai, acquired in December 2024 and integrated into the NVIDIA AI Enterprise stack [2], [3]. Run:ai adds GPU awareness to Kubernetes scheduling through fractional GPU allocation with memory enforcement, GPU pooling, dynamic MIG reconfiguration, topology-aware placement and workload-type-aware scheduling policies. The GPU awareness capabilities enhance GPU placement within Kubernetes. The capabilities do not model the operational state of the workloads Kubernetes places. Run:ai knows a pod requested one GPU with 40 GB memory and a node has a GPU available. Run:ai does not know the pod is running Granite 8B at 87% GPU utilization with 92% KV cache fill, P95 latency of 340 ms, queue depth of 12 and that these metrics carry semantic direction so the platform understands that higher KV cache fill means less capacity. Run:ai places workload, it is not an ontology.

NVIDIA acknowledged the limitation implicitly by building Dynamo (production release March 2026) and invested in llm-d as separate inference routing systems operating above Run:ai in the stack [4], [5]. The existence of these systems is an admission that Run:ai does not route inference requests.

The NVIDIA inference stack for enterprise deployments is therefore three products, Run:ai for container placement, Dynamo for inference routing and llm-d for distributed inference coordination. Three products, each addressing a gap the others leave, are bolted together through Kubernetes middleware.

The present paper proposes IBM Spectrum Symphony as the compute ontology extending insight into OpenShift and NVIDIA AI factory infrastructure. Symphony, a service-oriented dynamic compute platform descending from Songnian Zhou's 1987 doctoral research on dynamic load balancing [6], provides the intelligence layer Kubernetes lacks, delivering typed resource metrics, hierarchical governance, service lifecycle management and cross-substrate routing as aspects of one unified architecture rather than as separate products. Symphony's External Load Information Manager (ELIM) reports typed metrics with semantic direction from every resource type. Consumer hierarchies express organizational governance with priority, preemption, lending, borrowing and per-resource-group policy vectors. SOAM service graphs compose workloads across resource types with seven-phase service lifecycles and per-phase failure policies. The EGO resource broker unifies all resources under one workload management domain regardless of whether they reside on OpenShift, bare metal, cloud, or edge.

Symphony's compute ontology does not replace Kubernetes or OpenShift, it extends its perceptual reach into Kubernetes infrastructure, providing the intelligence layer the container platform lacks while Kubernetes retains full ownership of container runtime, networking, security and image lifecycle. The relationship parallels how Palantir Foundry's data ontology extends into data sources without replacing them [7], [8]. Foundry connects to databases, data lakes and streaming systems. The data sources continue operating. The ontology adds understanding and the ability to act on it. Symphony connects to OpenShift clusters, bare-metal hosts and cloud instances. The infrastructure continues operating. The ontology turns raw compute into actionable intelligence.

The contributions of the paper are sixfold. First, the paper formalizes the concept of a compute ontology as a typed, governed, semantically rich model of the compute domain, establishing the structural parallel to Palantir Foundry's data ontology and distinguishing the compute ontology categorically from container placement engines. Second, the paper provides a detailed comparison with Kueue, demonstrating that Kubernetes' strongest governance extension addresses one ontological dimension at alpha API maturity while leaving five dimensions entirely unaddressed. Third, the paper provides a feature-complete analysis of Run:ai v2.24 capabilities, demonstrating that every capability is replicable within Symphony's ELIM architecture and establishing that Run:ai exclusively uses public GPU APIs with no proprietary hardware access. Fourth, the paper identifies Spectrum Conductor on the same EGO resource broker as a fourth ontological layer providing GPU-accelerated data processing under the same governance framework. Fifth, the paper demonstrates that the compute ontology extends into OpenShift infrastructure through ELIM perception agents without modifying OpenShift's container runtime, networking, or security model. Sixth, the paper situates the compute ontology within the competitive landscape of NVIDIA's acquisitions of Run:ai and SchedMD, providing a vendor-independent alternative grounded in four decades of production deployment. A companion paper (Part II) presents empirical validation across three compute substrates.

II. The Compute Ontology

A. What a Compute Ontology Is

A compute ontology is a formal model specifying what compute resources exist, what typed properties those resources have, what relationships govern them, what operations apply to them and who controls what. Logic runs over the ontology and discovery follows from it. A resReq expression such as `select(vllm_kv_cache_usage_pct < 80) order(gpu_util)` evaluates typed resources with semantic direction to route a request to the optimal instance, demonstrating that the ontology is not merely descriptive but operational. The ontology models the compute domain the way a relational database models a data domain by defining the types, relationships and constraints governing the entries rather than merely storing them. A file system stores data. A relational database models the data domain. Kubernetes places containers. A compute ontology models the compute domain.

The distinction is categorical rather than incremental. A file system cannot become a relational database through feature additions because the two systems differ in what they represent, not in how fast they operate. A container placement engine cannot become a compute ontology through middleware extensions because the two systems differ in what they model, not in what they schedule. Kubernetes models container placement, namely what containers exist, what resources they requested and where they are. A compute ontology models compute intelligence, namely what resources are doing, how well they are doing it, what they should be doing and who governs what they do.

B. The Palantir Parallel

Palantir Foundry provides the data ontology for enterprise operations [7], [8]. Foundry maps digital assets to their real-world counterparts. A dataset of GPS coordinates becomes a fleet tracking system when the ontology defines Vehicle objects, Route links and PositionUpdate actions. The data existed before the ontology. The meaning did not. Foundry's Ontology Metadata Service defines object types with typed properties. Link types define relationships. Action types define operations with permissions and validation. AIP agents reason over the ontology. Every layer serves the ontology. The ontology is the center.

Palantir's connectors, the 150+ integrations feeding data into Foundry, are not the product. The ontology is the product. The connectors are perception. The connectors are how the ontology acquires knowledge of the data domain. No customer

adopts Palantir for its connectors. Customers adopt Palantir because the ontology makes their data intelligible and actionable in a way raw data infrastructure cannot.

Symphony provides the structural parallel for compute. Symphony's resource schema defines compute entities with typed properties. Consumer hierarchies define governance relationships. SOAM service lifecycles define operations with per-phase failure policies. The EGO resource broker unifies all resources under one workload management domain. ELIM is the perceptual apparatus of the compute ontology, the mechanism through which the ontology acquires knowledge of the compute domain. ELIM is not the product. The compute ontology is the product.

The parallel is structural, not metaphorical. Both systems solve the same problem in different domains by modeling a domain with typed entities, governed relationships and operational semantics so that the domain becomes intelligible and actionable rather than merely present.

C. The Zhou Lineage

The ontological character of Symphony is a direct inheritance from Songnian Zhou's 1987 doctoral research at UC Berkeley on dynamic load balancing in heterogeneous distributed systems [6]. Zhou's load index was not merely a scheduling input but a typed representation of host state with semantic meaning. The load index captured actual host state rather than a common abstraction and fed typed measurements into a scheduling algorithm whose placement decisions reflected what each host was doing. The implementation lineage is direct and unbroken, running from Zhou's load index (1987) [27] to Utopia (1988–1993) at the University of Toronto [9] to Platform LSF (1992) to Platform Symphony SOAM with ELIM (2006) to IBM Spectrum Symphony (2012–present) [10], [11], [12], [13].

ELIM generalizes Zhou's load index from per-host CPU metrics to per-type arbitrary metrics. Zhou's index reported CPU utilization and queue length for heterogeneous VAX and Sun workstations. ELIM can report `gpu_util`, `vllm_kv_cache_usage_pct`, `akida_power_mw` and `qpu_fidelity` for heterogeneous GPU, neuromorphic and quantum resources. The generalization preserved the original design, with each resource measured according to its own operational state, with all measurements feeding one scheduling algorithm through one metric interface. Critically, ELIM's contract is minimal, as any executable in any language, that writes typed metrics to stdout in ELIM's binary format participates in the compute ontology. The implementation can be a shell script calling nvidia-smi, a Python program using the Kubernetes API client, a Go binary with compiled performance matching native K8s operators, or a C/C++ program reading GPU memory maps through NVML at zero overhead. The perception depth of the ontology is bounded by the capabilities of the implementation language, not by ELIM's protocol. The principle scaled from VAX workstations to AI factories over 39 years because it was correct from the start and the polyglot implementation model ensures it scales into any platform domain the ontology needs to perceive.

Both LSF and Symphony share the same runtime infrastructure through the Dynamic Compute Platform (DCP), the same resource broker (EGO's VEMKD) and the same metric collection infrastructure (LIM/ELIM) [10], [11]. The shared infrastructure is architecturally significant because LSF and Symphony are not separate products forced into coexistence but complementary expressions of the same resource management framework applied to different workload paradigms. LSF applies Zhou's principle to batch workloads. Symphony applies it to service workloads. The multi-ontology architecture presented in prior work [14] leverages this complementarity for AI factory training and inference.

D. Symphony's Compute Ontology in Detail

Symphony's compute ontology operates across three layers corresponding to Foundry's three ontological layers.

Resource schema as type system. Symphony's `ego.shared` Resource block is a schema definition for the compute domain. Each resource declaration specifies a name, a data type (Boolean, Numeric, String), a reporting interval in seconds and a semantic direction indicating whether higher values represent more capacity or more load. The declaration `vllm_kv_cache_usage_pct Numeric 10 Y` creates a typed entity, an inference cache that exists, its fill level is numeric, reported every 10 seconds and higher values mean less available capacity. The declaration `gpu_mem_free Numeric 5 N` creates another, GPU memory availability is numeric, reported every 5 seconds and higher values mean more capacity. The platform understands the meaning of each metric because the metric was declared with its type and direction at registration time. Kubernetes resource requests are scalar quantities (CPU millicores, memory bytes, integer device counts) without types, semantic direction, or configurable reporting intervals.

ELIM scripts populate the schema with real-time values from every host and resource requirement expressions operate over the typed metrics. A resReq expression such as `select(vllm_model_name == "granite-8b" && vllm_kv_cache_usage_pct < 80) order(vllm_queue_depth)` is a query over the compute ontology selecting resources whose typed properties satisfy the expression and ordering them by a typed metric the platform understands. The reference implementation described in prior work reports over 100 unique resource metrics across seven categories [14], [15].

Consumer hierarchy as governance ontology. Symphony's consumer hierarchy is a tree-structured governance model expressing organizational intent in resource governance terms, operating across multiple governance dimensions simultaneously [10], [11]. Prioritization and preemption enable the platform to enforce organizational priorities when demand exceeds supply. Lending and borrowing ensure idle capacity flows laterally to siblings and vertically through the tree,

TABLE I
STRUCTURAL PARALLEL BETWEEN DATA AND COMPUTE ONTOLOGIES

Dimension	Foundry (Data Ontology)	Symphony (Compute Ontology)
Core question	What does our data mean?	What is our infrastructure doing?
Perception	150+ data connectors	ELIM typed metric reporting
Entity model	Object types with typed properties	Resource types with typed metrics and semantic direction
Governance	Role-based, marking-based, purpose-based access	Consumer hierarchies with priority, preemption, lending, borrowing
Relationships	Link types connecting objects	Service graphs composing workloads across resource types
Operations	Action types with permissions	SOAM 7-phase lifecycle with per-phase failure policies
Intelligence	AIP agents reason over ontology	resReq expressions query ontology; SOAM dispatchers act on it
Without it	Data in lakes, no meaning	Containers on nodes, no understanding
With it	Operational intelligence	Compute intelligence

with `EGO_DISTRIBUTION_INTERVAL=1` providing one-second rebalancing cycles. Multi-dimensional resource planning enables each consumer to hold independent resource policies per resource group. A single consumer can hold different share percentages, planned quotas and share quotas across GPU, CPU, neuromorphic and cloud resource groups simultaneously. Production Symphony deployments at major financial institutions manage thousands of consumers across deep hierarchy levels, processing billions of tasks daily [10], [11].

SOAM service lifecycle as operational semantics. SOAM's seven-phase service lifecycle (Register, CreateService, SessionEnter, SessionUpdate, Invoke, SessionLeave, DestroyService) with independent failure policies per phase captures the semantics of computational work rather than the mechanics of process management [10], [11]. Session types express distributed continuity guarantees. Proactive resource governance defines four boundary escalation levels (PROACTIVE, SEVERE, CRITICAL, HALT) for memory and virtual address space, preempting tasks before resource exhaustion rather than terminating them after. Multi-tier recursive task execution enables parent services to spawn child tasks across hosts within a unified session context. Multi-SSM failover provides transparent task-level redirection between physical service managers during failures.

Data processing as a fourth ontological layer. IBM Spectrum Conductor operates on the same EGO resource broker as Symphony, extending the compute ontology into data processing workloads [31]. Conductor provides Spark-on-EGO (Apache Spark executors scheduled as EGO tasks under consumer hierarchy governance), managed Jupyter notebook environments with Spark kernels, Anaconda/conda environment management shared across instance groups, Dask integration for distributed Python analytics and Deep Learning Impact for distributed training with hyperparameter search. Spark instance groups, notebook sessions and deep learning jobs participate in the same consumer hierarchy, the same ELIM metrics and the same GPFS coordination substrate as SOAM inference services. Multiple Spark versions run concurrently under different consumer policies. Combined with NVIDIA's cuDF acceleration on GPFS via GPUDirect Storage [32], the compute ontology governs GPU-accelerated data processing alongside inference, training and interactive exploration under one resource broker. Neither Run:ai nor Kueue provides any data processing scheduling, notebook management, or environment governance.

Together the four layers constitute a compute ontology, a typed and governed model of the compute domain that no container placement engine replicates.

III. THE KUBERNETES GAP

A. What Kubernetes Knows

Kubernetes answers four questions about the compute domain. First, what containers exist, since pod specifications declare containers with images, commands and environment variables. Second, what resources they requested, expressed as CPU millicores, memory bytes and GPU count declared at pod creation and static for the pod's lifetime. Third, where they are running, as node assignment is determined by the scheduler based on resource availability, affinity rules and taints. Fourth, whether they are alive, since liveness and readiness probes return binary health signals.

The four answers are placement facts. Placement facts address where something is and what it asked for. Placement facts do not address what something is doing, how well it is doing it, what it should be doing or who governs what it does.

B. What Kubernetes Cannot Express

Six categories of compute intelligence fall outside Kubernetes' architectural scope.

Typed metrics with semantic direction. ELIM metric declarations carry types (Boolean, Numeric, String), reporting intervals and semantic direction. The platform knows that `gpu_util 90` represents high load (direction Y) and `gpu_mem_free 40000` represents available capacity (direction N) without application-specific logic. Kubernetes Custom Metrics API adds metric sources but not typed metrics with declared semantic meaning.

Per-workload operational state. ELIM reports typed operational metrics from each service instance, with the number and nature of metrics defined by the entity itself. A vLLM inference instance exposes thirty-eight metrics [26], including KV cache fill, time to first token, time per output token,

queue depth, model identity, active sessions, prefix cache hit rate and their P95 variants [14], [15]. Kubernetes has no mechanism through which a running service instance can dynamically report custom operational metrics back to the resource broker in real time. The limitation is not merely that Kubernetes lacks specific metrics but that Kubernetes' architecture separates the scheduler from the workload. The Kubernetes scheduler makes a placement decision at pod creation and never receives operational feedback from the running workload. ELIM dissolves this separation because an ELIM executable running in the same execution context as the workload, implemented in any language capable of observing the workload's state, reports that state as typed metrics back to the resource broker continuously. A Python ELIM importing the vLLM metrics client has the same real-time visibility into inference state that vLLM's own internal monitoring has. A Go ELIM using client-go has the same access to Kubernetes API server state that any native operator has. The perception layer is as transparent as the implementation language allows and no language constraint limits what ELIM can observe.

Hierarchical organizational governance. Unlimited-depth consumer hierarchies with per-consumer, per-resource-group policy vectors. Priority, preemption, fairshare, lending and borrowing all rebalance every second. Kubernetes namespaces are flat labels with hard quota ceilings. Kubernetes has no hierarchical share propagation, no cross-namespace borrowing, no proportional entitlement, no per-resource-group policy vectors and no second-level rebalancing.

Request-level routing intelligence. SOAM dispatches individual inference requests to running service instances based on ELIM metrics, completing routing decisions in approximately 10 milliseconds [14]. Kubernetes Service routing operates at the connection level (round-robin or random) without workload awareness.

Cross-substrate governance. A single consumer hierarchy can govern OpenShift pods, bare-metal GPU hosts, cloud burst instances and neuromorphic edge nodes simultaneously. Kubernetes operates within a single cluster boundary. Cross-cluster routing requires external service mesh infrastructure.

Service lifecycle depth. Seven-phase SOAM lifecycle manages per-phase failure policies, session continuity across client disconnects, proactive boundary escalation and multi-tier recursive task execution. Kubernetes pods have one lifecycle (pending, running, succeeded, failed), one failure action (restart or not), no session continuity, no per-phase recovery, no proactive boundary escalation and no multi-tier recursive task context.

C. The Kueue Comparison

Kueue (v0.17, Kubernetes SIG) [30] is the strongest Kubernetes-native candidate for closing the governance dimension of the gap. A detailed comparison demonstrates that Kueue approaches one dimension of the compute ontology at alpha quality while leaving five dimensions entirely unaddressed.

Hierarchy. Symphony consumer hierarchies support unlimited depth with production API maturity, modeling organizational structures at any level of granularity. Kueue provides hierarchical Cohorts via a `spec.parent` field on the Cohort CRD, permitting arbitrary nesting. The Cohort CRD is v1alpha1 with known defects, as GitHub Issue #3948 documents broken `reclaimWithinCohort` preemption with hierarchical cohorts. Symphony's consumer hierarchy has operated at production scale for decades across major financial institutions managing thousands of consumers.

Lending and borrowing. Symphony's idle capacity flows automatically through the consumer tree, laterally to siblings and vertically through the hierarchy, governed by per-consumer per-resource-group share percentages and reclaimed within one second when the owning consumer has demand. Kueue provides `lendingLimit` and `borrowingLimit` per ClusterQueue per Resource Flavor, limiting how much unused quota others can borrow and how much a queue can borrow beyond nominal. Kueue prioritizes non-borrowing workloads over borrowing ones but has no mechanism for automatic capacity flow through a hierarchy.

Rebalancing. Symphony rebalances every second via `EGO_DISTRIBUTION_INTERVAL=1`, periodically redistributing shares across the consumer tree as demand changes. Already-running workloads are subject to rebalancing. Kueue operates on event-driven reconciliation with no periodic rebalancing sweep. Already-admitted workloads are never redistributed. Preemption triggers only when a new pending workload arrives. Cooperative preemption enabling graceful checkpoint-based preemption is listed as a 2026 priority and is not yet available. The absence of periodic rebalancing means that a Kueue cluster with an organizational priority shift cannot reclaim resources from already-running workloads until a new workload submission triggers the preemption algorithm.

Preemption. Symphony preempts at the task level with graduated escalation across four boundary levels (PROACTIVE, SEVERE, CRITICAL, HALT), preserving other tasks and sessions within a service while preempting the specific task consuming excess resources. Proactive boundary management preempts before resource exhaustion rather than terminating after. Kueue preempts at the workload level, evicting all pods in a workload as a unit. Three policy axes control preemption, `withinClusterQueue`, `reclaimWithinCohort` and `borrowWithinCohort`. Fair sharing preemption strategies (`LessThanOrEqualToFinalShare`, `LessThanInitialShare`) constrain when preemption is permitted. Kueue's workload-level preemption is coarser than Symphony's task-level preemption but finer than Kubernetes' pod-level restart and the fair sharing strategies prevent pathological preemption cascades.

Per-resource-group policy vectors. A single Symphony consumer holds independent policy vectors (share percentage, planned quota, hard quota, priority, preemption rules, lending policies) across GPU, CPU, neuromorphic, cloud burst and other resource groups simultaneously. One consumer can hold

40% of GPU resources, 60% of CPU, 20% of neuromorphic and 10% of cloud burst capacity under different governance rules per group. Kueue's Resource Flavors allow per-flavor quotas within a ClusterQueue but the Flavor model maps to hardware classes via node labels rather than to organizational governance dimensions.

Five dimensions Kueue does not address. Kueue is an admission controller governing when workloads start. It does not perceive running workload state (no per-workload operational metrics, no metric freshness concept, no semantic direction). It does not route requests (no inference routing, no semantic routing, no session affinity, no KV cache awareness). It does not manage service lifecycle (no per-phase failure policies, no session continuity, no proactive boundary escalation, no multi-SSM failover). It does not coordinate across substrates (MultiKueue provides static multi-cluster dispatch with significant limitations including no ResourceClaim synchronization, no manager-as-worker and no quota consistency across clusters). It does not manage data (no coordination substrate, no model artifact management, no KV cache coordination).

The comparison demonstrates that Kueue addresses one dimension of the compute ontology (hierarchical governance) at alpha maturity with known defects, without periodic rebalancing and without per-resource-group policy depth matching Symphony's consumer model. The remaining five dimensions of the compute ontology (perception, routing, lifecycle, cross-substrate scaling, coordination) fall entirely outside Kueue's scope because Kueue is an admission controller, not a compute ontology.

D. Why Extensions Do Not Close the Gap

Kubernetes-native extensions beyond Kueue address specific dimensions of the gap without closing it. Custom Resource Definitions add data structures but not typed metrics with semantic direction. The Custom Metrics API and Prometheus add metric sources but Prometheus operates at 15–30 second default scrape intervals (configurable to 1 second with resource cost) and requires a middleware chain (Prometheus scraper, Custom Metrics Adapter, KEDA or HPA) traversing four or five software boundaries from metric production to scaling action. An ELIM executable reports typed metrics directly to the resource broker through one protocol hop. KEDA provides event-driven autoscaling but not request-level routing. Knative provides serverless serving but not workload-aware routing based on real-time operational state. The Gateway API (GA in Kubernetes 1.29) provides weighted and header-based routing at the HTTP level but not workload-aware routing based on KV cache fill, queue depth, or model-tier classification.

Each extension addresses one dimension. The extensions do not compose into a compute ontology because they share no type system, no governance hierarchy, no unified metric protocol and no cross-substrate scheduling domain. Symphony is the typed, governed, semantically rich model itself, not a composition of independent middleware.

IV. Run:ai Analysis

A. Run:ai Capabilities (v2.24)

Run:ai provides GPU-aware scheduling within Kubernetes [2]. The capability inventory includes fractional GPU allocation with runtime memory enforcement through proprietary GPU fractions logic (with dynamic fractions supporting burst between a guaranteed request and a limit), GPU memory swap extending usable GPU memory beyond physical limits, GPU pooling with per-department and per-project quotas per node pool, MIG support via profiles (though MIG and GPU fractions cannot coexist on the same node), topology-aware placement considering NVLink, NVSwitch and MNNVL interconnects across a five-level hierarchy (region, zone, block, rack, hostname) including GB200 NVL72 support, gang scheduling for multi-GPU distributed training, bin packing, spread and consolidation scheduling policies, Dominant Resource Fairness with time-based fairshare tracking historical GPU-hour consumption, NIM-native autoscaling with configurable metrics (latency, throughput, concurrency) and scale-to-zero, Knative integration for single-node serverless inference, DynamoGraphDeployment for disaggregated prefill/decode inference pipelines (v2.23+) and GPU utilization dashboards with Prometheus export.

Run:ai's multi-tenancy operates across four levels, Tenant, Cluster, Department and Project. Departments group projects and receive GPU/CPU quotas per node pool with scheduling rules (idle GPU timeout, workload time limits, node affinity). Policy rules cascade with tenant rules taking precedence over cluster rules over department rules over project rules. The hierarchy does not support nesting departments within departments; the organizational model is fixed at four levels.

NVIDIA open-sourced the underlying KAI Scheduler under Apache 2.0 in April 2025, making the scheduling engine available independently of the commercial platform [16]. The open-source KAI Scheduler provides gang scheduling, bin-packing, topology-aware scheduling, hierarchical queues (two-level, department and project), DRF and time-based fairshare. The commercial platform retains fractional GPU memory isolation, GPU memory swap, the full multi-tenant control plane, inference workload management (Knative integration, NIM templates, autoscaling), dashboards, analytics and the policy engine [16].

B. Run:ai as Placement Engine, Not Ontology

Run:ai enhances Kubernetes' placement decisions with GPU awareness. However, the enhancement does not constitute a compute ontology because Run:ai operates at the same categorical level as Kubernetes and decides which node gets a pod. After placement, Run:ai's visibility ends. Run:ai does not know what model the pod is serving, how full the KV cache is, what the P95 latency looks like, whether the next request should go to this instance or another, or whether this instance should yield resources to a higher-priority consumer based on real-time operational state.

The distinction is between knowing a container has one GPU allocated and knowing the GPU is at 87% utilization

TABLE II
KUEUE GOVERNANCE COMPARED TO SYMPHONY COMPUTE ONTOLOGY

Dimension	Symphony	Kueue (v0.17)
Hierarchy depth	Unlimited, production API	Arbitrary, **alpha API (v1alpha1), known bugs**
Lending/borrowing	Automatic flow through tree, 1s reclaim	Per-flavor limits, no automatic flow
Rebalancing	Periodic (1 second), redistributes running workloads	**Event-driven only, no periodic sweep, no redistribution of running workloads**
Preemption granularity	Task-level with 4-level proactive escalation	Workload-level (all pods evicted as unit)
Per-resource-group policies	Independent policy vectors per consumer per RG	Per-flavor quotas within ClusterQueue
Fairshare	Proportional share with historical decay, 1s cycle	DRS for preemption; Admission Fair Sharing (alpha) with configurable decay
Request routing	Per-request, 38-metric scoring, ~10 ms	**None**
Workload perception	38+ typed metrics per instance, sub-second	**None (admission controller only)**
Service lifecycle	7-phase SOAM, per-phase failure policies	**None (suspended/unsuspended gate only)**
Session management	Multi-SSM failover, session continuity	**None**
Multi-cluster	Overflow: policy/threshold/session/task-based	MultiKueue: admission-check-based dispatch, alpha, significant limitations
Hardware platforms	NVIDIA, AMD, Intel, neuromorphic, quantum, mainframe	Hardware-agnostic via node labels (no accelerator awareness)
Coordination substrate	GPFS: RDMA, xattrs, ILM, GPUDirect Storage	**None**

with 92% KV cache fill and a P95 latency of 340 ms. The distinction is between container scheduling and compute intelligence.

Run:ai collects approximately 30 standard cluster metrics, 12 GPU profiling metrics through DCGM and 11 NIM-specific inference metrics through Prometheus at 15–30 second scrape intervals [2]. The NIM-specific metrics (TTFT, KV cache usage percentage, request counts) are sourced from the NIM runtime itself, not collected or interpreted by Run:ai's scheduler. Run:ai does not use these metrics for scheduling or routing decisions. The metrics serve observability dashboards, not the scheduling loop. The distinction is critical because having metrics and using metrics for scheduling are different capabilities. Run:ai has inference metrics available for human observation. Symphony's compute ontology has inference metrics feeding the scheduling and routing algorithms at sub-second intervals.

Run:ai's claimed "NVIDIA hardware integration" warrants examination. Every GPU API Run:ai uses is publicly available to any software. NVML ships with the GPU driver and is documented for third-party use. DCGM is open source under Apache 2.0. CUDA runtime, Unified Virtual Addressing, MPS, MIG and time-slicing are public CUDA features configurable by any Kubernetes scheduler through the NVIDIA GPU Operator. Run:ai's fractional GPU memory enforcement, the one component not open-sourced with the KAI Scheduler, uses standard CUDA call interposition techniques (intercepting cudaMalloc via LD_PRELOAD or a custom device plugin shim), a well-documented approach available to any engineering team. Run:ai's GPU memory swap uses CUDA Unified Virtual Addressing, a public CUDA feature available since CUDA 4.0. The "NVIDIA AI Enterprise" certification is a commercial software license bundle, not a technical capability granting exclusive API access. The "DGX-Ready Software" designation is a validated partner program confirming tested configurations and support agreements. ELIM reads the same NVML and DCGM interfaces Run:ai uses, reporting GPU state as typed metrics with semantic direction feeding the scheduling and routing algorithms directly.

Run:ai's architectural limitations extend beyond the ontological gap. Run:ai's full feature set (fractional GPUs, GPU profiling, NIM integration, GPU memory swap) requires NVIDIA hardware and DCGM; AMD support is limited to DRA ResourceClaims in the open-source KAI Scheduler without fractional GPU or GPU profiling capabilities and Intel support is absent. Run:ai requires Kubernetes infrastructure, preventing management of bare-metal HPC clusters or non-containerized workloads. Run:ai's control plane manages multiple clusters for policy, RBAC and dashboards but scheduling operates strictly within a single Kubernetes cluster. There is no cross-cluster workload scheduling, no cross-cluster resource federation and no ability to schedule a single workload across nodes in different clusters. Within a cluster, scheduling does not cross node pool boundaries automatically; users submit a prioritized list of node pools and the system tries them sequentially. Run:ai's four-level hierarchy (Tenant, Cluster, Department, Project) with no department nesting cannot express the deep organizational hierarchies financial institutions or the large enterprise environments require.

C. Feature-by-Feature Replication in Symphony

Every Run:ai capability is replicable within Symphony's ELIM architecture with equivalent or superior depth.

Fractional GPU allocation. ELIM reports per-GPU memory state (`gpu_mem_used`, `gpu_mem_free`, `gpu_mem_total`). EGO's resource broker allocates GPU memory ranges to consumers through resReq expressions (`select(gpu_mem_free >= 20000)`). Run:ai enforces fractional memory limits through CUDA call interposition, intercepting cudaMalloc via LD_PRELOAD or a custom device plugin shim to refuse allocations exceeding the fraction. The technique is effective but uses standard CUDA interfaces available to any scheduler. Symphony's proactive boundary management adds graduated escalation (PROACTIVE at

50%, SEVERE at 40%, CRITICAL at 25%, HALT at 15%), preempting tasks before exhaustion rather than enforcing only at the allocation boundary. The distinction is between enforcement at the point of allocation and governance across the full memory lifecycle.

GPU pooling. EGO resource groups provide GPU pooling with per-consumer, per-resource-group policies. The same physical GPU host participates in multiple resource groups (`gpu_rg`, `vllm_gpu_rg`, `pqfm_gpu_rg`) with different slot counts and governance policies. A single A100 GPU can serve general compute, vLLM inference and quantum feature encoding simultaneously under different governance per group.

Dynamic MIG reconfiguration. ELIM reports MIG partition state per GPU (`gpu_mig_mode`, `gpu_mig_profile`, `gpu_mig_available`) through nvidia-smi or NVML queries. resReq expressions allocate MIG instances based on workload requirements (`select(gpu_mig_profile == "3g.40gb" && gpu_mig_available > 0)`). Run:ai supports MIG via profiles but MIG and GPU fractions cannot coexist on the same node, limiting deployment flexibility. ELIM's MIG reporting carries no such constraint because ELIM reports state without imposing scheduling-mode restrictions.

Topology-aware placement. ELIM reports per-GPU interconnect type and bandwidth through topology-aware metric scripts, enabling resReq expressions to select GPUs based on NVLink connectivity and bandwidth characteristics across clusters. Run:ai provides topology awareness within a five-level hierarchy (region, zone, block, rack, hostname) including NVLink, NVSwitch and MNNVL support but confined to a single Kubernetes cluster. ELIM topology metrics span all substrates under one scheduling domain.

Gang scheduling. EGO provides atomic multi-resource allocation across the scheduling domain, proven with LSF at supercomputer scale (27,648 GPUs across 4,608 nodes at Oak Ridge Summit [18]). Run:ai's gang scheduling operates within the single-cluster boundary.

Multi-tenant governance. Symphony consumer hierarchies provide unlimited-depth organizational modeling with per-consumer, per-resource-group policy vectors, compared to Run:ai's two-level Department/Project model with namespace quotas.

Inference autoscaling. Symphony's ELIM-informed scaling operates on 38+ real-time operational metrics with sub-second freshness, compared to Run:ai's NIM-native autoscaling operating on Prometheus-scraped metrics at 15–30 second intervals.

D. Capabilities Beyond Run:ai

The following capabilities arise from Symphony's ontological depth and have no Run:ai equivalent because they require a typed resource model, governance hierarchies, service lifecycle semantics, or cross-substrate coordination that no Kubernetes-based GPU placer can express.

Request-level inference routing. SOAM dispatches individual inference requests to running vLLM [25], [26] instances based on ELIM metrics, with four weighted scoring algorithms (LoadAwareScorer at 0.40, SessionAffinityScorer at 0.25, NoHitLRUScorer at 0.20, ActiveRequestScorer at 0.15) completing the routing decision in approximately 10 milliseconds [14]. Run:ai places pods. Symphony routes requests.

Semantic model-tier routing. A KNN classifier trained on 415 samples with MiniLM embeddings achieves 91.1% accuracy across code, math and general domains, routing queries to appropriate model tiers through Symphony's native resReq expressions (`select(llm_score_code >= 70) order(llm_score_code)`) [14], [15]. Cost savings of 74–81% are achieved by routing simple queries to smaller models rather than dispatching all queries to the largest model [19], [20].

KV cache-aware routing and cross-model transfer. ELIM reports `vllm_kv_cache_usage_pct` and `vllm_prefix_cache_hit_rate` per instance. GPFS-based three-tier KV cache architecture enables cross-model transfer at 340 ms for 4,096-token contexts, compared to 2.8 seconds for full context recomputation, an 8.2x improvement [14].

Cross-cluster federation and scaling architecture. Symphony Overflow provides policy-based multi-cluster execution with configurable thresholds and session-based and task-based routing granularity. Workloads route across clusters based on ELIM metrics from all sites, governed by the same consumer hierarchy spanning all clusters. HostFactory provisions cloud GPU instances (AWS, Azure, GCP, IBM Cloud) into the same scheduling domain, governed by the same consumer hierarchy and ELIM metrics as on-premise resources. The scaling architecture is horizontally unbounded because adding a new cluster, cloud, site, or accelerator type extends the workload management domain without architectural modification [10], [11].

Run:ai's control plane manages multiple clusters for policy, RBAC and dashboards but scheduling operates strictly within one Kubernetes cluster. There is no cross-cluster workload scheduling. Within a cluster, Run:ai inherits the Kubernetes cluster size ceiling (typically 5,000–15,000 nodes depending on Kubernetes version and etcd configuration). Run:ai's scaling model is vertical within one cluster on one GPU vendor. Symphony's scaling model is horizontal across clusters, clouds, sites and hardware platforms under unified governance.

Heterogeneous accelerator support. ELIM reports typed metrics from NVIDIA, AMD, Intel, BrainChip neuromorphic and IBM Quantum resources through the same protocol [15]. Run:ai's full feature set (fractional GPUs, GPU profiling, NIM integration, GPU memory swap) requires NVIDIA hardware and DCGM. AMD support is limited to DRA ResourceClaims in the open-source KAI Scheduler without fractional GPU or GPU profiling capabilities. Intel support is absent.

GPFS coordination. Model artifacts, KV cache state, training checkpoints, inference telemetry and model readiness signaling through extended attributes flow through GPFS

TABLE III
RUN:AI CAPABILITIES REPLICATED AND EXCEEDED IN SYMPHONY

Capability	Run:ai (v2.24)	Symphony ELIM	Advantage
Fractional GPU	Memory enforcement (proprietary)	Proactive boundary management (4 levels)	Prevents OOM vs reacts
GPU pooling	Per-department quota	Per-consumer per-resource-group vectors	Multi-dimensional governance
Dynamic MIG	MIG profiles (no coexistence with fractions)	ELIM MIG state reporting (no constraints)	No mode restriction
Topology awareness	NVLink/NVSwitch/MNNVL, single cluster	ELIM topology metrics, cross-substrate	Multi-cluster, proven at 27K GPUs
Gang scheduling	All-or-nothing, single cluster	EGO atomic allocation, cross-substrate	Proven at supercomputer scale
Multi-tenancy	4 levels (Tenant/Cluster/Dept/Project), no nesting	Unlimited depth consumer hierarchy	Organizational digital twin
Preemption	Pod-level	Task-level with graduated escalation	Preserves sessions, not just pods
Fair sharing	Over-quota borrowing	Lending/borrowing with 1s rebalancing	15x faster rebalancing
Inference scaling	NIM-native (Prometheus)	ELIM 38 metrics (sub-second)	Sub-second metric freshness

with RDMA transport, ILM tiering and quorum-based consistency [14]. Run:ai has no coordination substrate. It relies on Kubernetes-native storage classes (PVC, S3, NFS) with no cross-workload coordination mechanism, no model artifact management and no KV cache coordination across inference instances. Dynamo provides KV cache offloading across memory hierarchies and NIXL data transfer but these are Dynamo capabilities, not Run:ai capabilities.

V. ARCHITECTURE

A. Extending the Compute Ontology into OpenShift

When Palantir connects to a new data source, Foundry extends its ontological reach. The data source becomes part of what the ontology perceives. Objects in the ontology reflect the state of the data source. The data source does not change. The ontology's understanding grows.

When Symphony extends into an OpenShift cluster, the compute ontology extends its perceptual reach. The OpenShift GPU nodes become part of what the ontology perceives. Resource metrics in the ontology reflect the operational state of every workload on those nodes. OpenShift does not change. The ontology's understanding grows.

The mechanism is ELIM, the perceptual apparatus of the compute ontology. ELIM perception agents on OpenShift GPU nodes report typed operational metrics to Symphony's LIM infrastructure. The agents are deployed as a DaemonSet on GPU node pools, providing host-level GPU metrics through nvidia-smi and DCGM, workload-level inference metrics through vLLM's metrics endpoint and node-level system metrics through standard system interfaces. MELIM (Master ELIM) on each host merges multiple ELIM metric streams, validates syntax and forwards consolidated data to LIM, which reports to Symphony's master LIM and EGO resource broker. The metric flow is identical to bare-metal Symphony; the only difference is that ELIM scripts execute inside containers rather than directly on hosts.

ELIM's perceptual depth into Kubernetes is not constrained by the protocol. ELIM is a site-definable executable that collects metrics and writes them to stdout in a defined binary format. Any program that can write to stdout can be an ELIM, including shell scripts, Python programs, C/C++ binaries, Go executables, Rust programs. The choice of implementation language determines the depth of perception available to the compute ontology.

A Python ELIM using the `kubernetes` client library has full Kubernetes API server access. The ELIM can watch pod lifecycle events in real time, query the scheduler's binding decisions, read Custom Resource Definitions including Run:ai's GPU allocation CRDs, inspect Horizontal Pod Autoscaler state, read Prometheus TSDB directly for historical metric context, monitor etcd health and track namespace resource quota consumption. A Go ELIM using `client-go` has the same depth as any native Kubernetes operator or controller, with the additional advantage of compiled performance and direct access to the informer cache. A C/C++ ELIM can read GPU memory maps through CUDA or NVML with zero marshaling overhead, or interface with DCGM's shared memory structures for sub-millisecond GPU metric capture.

The architectural significance is that the compute ontology is not an external observer looking at Kubernetes through a narrow metrics endpoint. The ELIM perception layer operates with the same programmatic access to Kubernetes internals as any operator, controller, or admission webhook. The compute ontology sees everything Kubernetes sees. It also sees everything the workloads see (vLLM inference state, GPU operational metrics, training checkpoint state). The ontology adds what neither Kubernetes nor the workloads provide, namely typed semantics, semantic direction, hierarchical governance and cross-substrate awareness. The ontology's perceptual reach into the platform is as deep as the platform allows any native program to go.

The perceptual transparency extends to Run:ai. A Python ELIM can query Run:ai's REST API for GPU allocation state, fractional GPU assignments, department quota utilization and workload scheduling decisions. The same ELIM reports Run:ai's internal state as typed metrics in the compute ontology alongside vLLM inference metrics and GPU hardware metrics. The compute ontology perceives Run:ai as one more data source feeding its understanding of the compute domain, not as a competing scheduler to circumvent.

TABLE IV
CAPABILITIES BEYOND RUN:AI

Capability	Run:ai	Symphony
Inference routing granularity	Pod (placement)	Request (38-metric routing)
Metric freshness	15–30s (Prometheus)	Sub-second (ELIM push)
Inference metrics for scheduling	Available for dashboards, not used by scheduler	Feed scheduling and routing algorithms directly
Semantic routing	Not available	resReq expressions, 91.1% accuracy
KV cache-aware routing	Not available	ELIM `vllm_kv_cache_usage_pct`
Cross-model KV transfer	Not available (Dynamo provides separately)	GPFS-mediated, 340 ms / 4K tokens
Cross-cluster scheduling	Not available (centralized management only)	Overflow: policy/threshold/session/task-based
Cloud bursting	Not available	HostFactory into same scheduling domain
Scaling ceiling	Single K8s cluster (5–15K nodes)	Unbounded (add clusters/clouds/sites)
Heterogeneous accelerators	NVIDIA (full), AMD (limited DRA), Intel (none)	Any resource type via ELIM
Service lifecycle	Pod restart	SOAM 7-phase, per-phase failure policies
Session management	None	Multi-SSM failover, session continuity
Inference cost optimization	Not available	74–81% savings via semantic routing
Model handoff (train→inference)	External pipeline	GPFS xattr-mediated, 47s automatic
Coordination substrate	K8s storage pass-through	GPFS: RDMA, xattrs, ILM, GPUDirect Storage
Data processing	None	Spark-on-EGO, Jupyter, conda, Dask under same consumer hierarchy
GPU API exclusivity	None (all public NVML/DCGM/CUDA APIs)	Same public APIs + GPUDirect Storage on GPFS
Dispatch overhead	~100 ms (pod schedule)	~10 ms (ELIM-based routing)

OpenShift retains full ownership of container runtime (containerd/CRI-O), networking (CNI plugins, service discovery, ingress, network policy), security (RBAC, Pod Security Standards, service mesh mTLS), storage (CSI drivers, PersistentVolumeClaims), CI/CD (ArgoCD, Tekton) and node management (kubelet, node lifecycle, machine health). The compute ontology adds understanding. It does not modify infrastructure. But its understanding is as deep as the infrastructure exposes, because ELIM's implementation language determines perception depth and no language constraint limits what ELIM can observe.

B. Three Modes of Ontological Engagement

Three engagement modes accommodate different operational contexts.

Mode 1: Ontology Enrichment. The compute ontology extends directly into the OpenShift cluster. ELIM perception agents deploy on GPU nodes via the Symphony Operator. Consumer hierarchies govern OpenShift projects with full hierarchical depth. SOAM dispatches inference requests based on ELIM metrics. The ontology is native to the cluster. Mode 1 provides the deepest integration and is the recommended approach for enterprises committed to OpenShift as their platform.

Mode 2: Ontology Governance. The compute ontology governs the OpenShift cluster from outside. ELIM perception agents report to Symphony externally. The ontology makes routing and scaling decisions, executing through the Kubernetes API. Mode 2 requires zero modification to the OpenShift cluster and is appropriate for regulatory environments where OpenShift certification must remain untouched.

Mode 3: Ontology Subsumption. The compute ontology governs Run:ai as one typed resource within its domain. Run:ai handles GPU fractioning and container placement. ELIM perception agents add the 38 inference metrics Run:ai cannot see. Symphony makes routing decisions based on the full metric space. Run:ai executes container operations. Mode 3 neutralizes the "Run:ai versus Symphony" framing by layering Symphony's ontological depth above Run:ai's GPU placement and is appropriate for enterprises with existing Run:ai deployments.

The three modes compose. An enterprise runs Mode 1 on their primary OpenShift cluster, Mode 2 on a secondary cluster, Mode 3 on a Run:ai-managed cluster and adds bare-metal DGX hosts, cloud burst instances and neuromorphic edge nodes, all under one compute ontology, one consumer hierarchy, one set of ELIM metrics, one SOAM routing framework.

C. Consumer Hierarchy Mapping

Symphony consumer hierarchies map onto OpenShift projects with richer governance.

- Hierarchical share propagation, where a consumer owning 5% of a resource group can use 100% if no siblings have demand, yielding within one second when demand returns. OpenShift ResourceQuotas are hard ceilings.
- Cross-namespace borrowing, where development consumers borrow from idle production capacity. OpenShift has no cross-namespace resource sharing.
- Priority-based preemption, where GPU demand exceeds supply, the consumer hierarchy redistributes shares per organizational priority. Preemption operates at the task level rather than the pod level, preserving inference sessions rather than terminating entire containers.
- Per-resource-group policies, where the same consumer holds different share percentages for GPU, CPU, neuromorphic and cloud resource groups simultaneously.

- Sub-second rebalancing, with `EGO_DISTRIBUTION_INTERVAL=1` rebalancing every second. Kubernetes HPA default is 15 seconds.

D. SOAM Routing on OpenShift

Symphony's SOAM dispatcher routes inference requests to OpenShift-hosted vLLM instances using ELIM-informed scoring algorithms. The dispatcher evaluates ELIM metrics from all available instances across all substrates and selects the optimal target. The Kubernetes service mesh handles network routing. Symphony handles workload routing. The service mesh sends the request to a backend, while Symphony sends the request to the right backend.

The distributed inference deployment implements four scoring algorithms as native SOAM services [14], LoadAwareScorer (weight 0.40, using `vllm_queue_depth` with GPU utilization penalty above 90%), SessionAffinityScorer (weight 0.25, header-based sticky routing with LRU cache), NoHitLRUScorer (weight 0.20, distributing cold requests evenly) and ActiveRequestScorer (weight 0.15, using `vllm_active_sess`). The combined routing decision completes in approximately 10 milliseconds.

E. GPFS as Coordination Substrate

GPFS connects the compute ontology across all substrates through a unified storage fabric [14], [21]. Model weights, KV cache state, training checkpoints, inference telemetry and model readiness signaling through extended attributes (xattrs) flow through GPFS with RDMA transport, ILM tiering and quorum-based consistency. GPUDirect Storage enables direct data transfer between GPU memory and GPFS without CPU intermediation [22]. The same GPFS fabric serving bare-metal GPU hosts serves OpenShift-hosted workloads through GPFS CSI drivers, providing a single coordination substrate spanning all compute substrates.

NVIDIA validated this architectural choice at GTC 2026 by selecting IBM Storage Scale (GPFS) to provide 10 petabytes of high-performance storage for GPU-native analytics engines, certified on NVIDIA DGX platforms [22], [34]. The IBM-NVIDIA collaboration announced at GTC 2026 integrates NVIDIA cuDF acceleration with IBM watsonx.data's Presto SQL engine through the Velox execution engine, delivering 5x query speedup and 83% cost reduction in production proof-of-concept testing [33]. NVIDIA's cuDF and RAPIDS libraries access GPFS through GPUDirect Storage via the cuFile API and kvikIO [32], loading data directly into GPU DataFrames from GPFS without CPU intermediation. NVIDIA's own GPU-accelerated data processing stack runs on the same coordination substrate the compute ontology uses. Run:ai has no storage layer, no data processing capability and no equivalent to GPUDirect Storage.

F. Cross-Substrate Routing

The compute ontology evaluates ELIM metrics from all substrates simultaneously when making routing decisions. An inference request arriving at the SOAM dispatcher is evaluated against metrics from OpenShift-hosted vLLM instances, bare-metal vLLM instances and cloud burst instances in one scoring pass. The routing decision considers KV cache fill, queue depth, P95 latency, GPU utilization, model tier and consumer governance policies regardless of where the instance resides. HostFactory provisions cloud GPU instances when demand exceeds local capacity. Overflow routes across geographically distributed clusters.

The cross-substrate capability is architecturally significant because it eliminates the single-cluster boundary Kubernetes and Run:ai impose. An enterprise running the compute ontology can route inference requests to the optimal instance across on-premise OpenShift, on-premise bare metal and cloud burst or overflow capacity under unified governance.

VI. Validation Context

A. Prior Demonstrations

The compute ontology's core capabilities are established through 25+ demonstrations across three companion papers. The multi-ontology AI factory paper [14] demonstrates ELIM-informed inference routing with four weighted scoring algorithms achieving approximately 10 ms routing decisions, semantic model-tier routing with 91.1% accuracy, KV cache-aware routing with GPFS-mediated cross-model transfer at 340 ms for 4,096-token contexts (8.2x improvement over recomputation), consumer hierarchy governance with sub-second rebalancing and cost savings of 74–81% through semantic routing. The quantum-centric heterogeneous orchestration paper [15] demonstrates ELIM accommodation of resources differing in kind across QPU, NPU, GPU, CPU and mainframe tiers. The Palantir sovereign AI OS paper [8] demonstrates Symphony as a compute ontology parallel to Foundry's data ontology with near-full-spectrum neuromorphic perception.

B. OpenShift Validation Design (Part II)

A companion paper (Part II) will present empirical validation of the compute ontology on OpenShift infrastructure across three compute substrates: managed OpenShift on IBM Cloud (ROKS) with 1–2 A100 GPU worker profiles, self-managed OpenShift on commodity hardware (AMD EPYC 7702P, five NVIDIA RTX 3090 GPUs with GPU passthrough) and a bare-metal host with eight NVIDIA A100 (80 GB) GPUs on the same IBM Cloud VPC. The ROKS GPU density limitation is architecturally significant: managed Kubernetes GPU profiles across cloud providers generally offer lower GPU-per-node density than bare-metal DGX systems, making cross-substrate routing operationally necessary for multi-tier model serving spanning 2B to 34B parameters. Five demonstrations validate the architecture: ontological perception of OpenShift workloads, ELIM-informed inference routing versus Kubernetes round-robin, Istio/Envoy service mesh and Run:ai baselines, cross-substrate routing across all three substrates, consumer hierarchy governance with sub-second rebalancing across substrates and semantic routing with model-tier escalation and GPFS-mediated KV cache transfer across

substrate boundaries. The Symphony Operator for OpenShift deploys ELIM perception agents as a DaemonSet on GPU node pools.

VII. Evaluation

A. Ontological Depth

The evaluation framework measures ontological depth across three dimensions rather than isolated performance metrics.

Perception depth. Kubernetes perceives 3 static scalar quantities per workload (CPU request, memory request, GPU count). Run:ai perceives GPU count, MIG profile and fractional allocation. The compute ontology perceives 38+ typed dynamic metrics per instance with semantic direction at subsecond freshness. The perception gap is both quantitative and structural. The current ELIM configuration reports 38+ typed dynamic metrics per vLLM instance, but the dimensionality is not fixed. ELIM's metric space is defined by the nature of the resource being perceived, making it extensible without architectural modification. Each additional metric expands the combinatorial space of decisions the ontology can make, from resource selection through resReq compositions to routing algorithms, consumer rebalancing, proactive boundary escalation and cross-substrate evaluation, providing exponential growth in actionable visibility as the metric set grows. Kubernetes' 3 static scalar quantities are fixed by the API.

Governance depth. Kubernetes provides 1-level namespace quotas with hard ceilings. Kueue provides hierarchical Cohorts with borrowing and preemption at alpha API maturity (v1alpha1) with known defects, event-driven reconciliation with no periodic rebalancing and workload-level preemption triggered only by new pending workloads. Run:ai provides a 4-level hierarchy (Tenant/Cluster/Department/Project) with no department nesting, over-quota borrowing via weights and Dominant Resource Fairness. Symphony's compute ontology provides unlimited-depth consumer hierarchies with per-consumer, per-resource-group policy vectors, one-second periodic rebalancing of already-running workloads, task-level preemption with proactive 4-level escalation and automatic capacity flow through the governance tree. The governance gap is structural, Kueue and Run:ai provide quota management. Symphony's compute ontology provides organizational modeling.

Routing depth. Kubernetes routes at the connection level (round-robin/random). Run:ai does not route requests. Symphony routes at the request level with multiple metrics scoring, semantic model-tier classification, KV cache awareness and cross-substrate evaluation. The routing gap is categorical because Run:ai does not route requests at all.

B. Performance Context

C. Scalability

The architecture inherits scaling properties from its constituent platforms. A single Symphony cluster supports 5,000 compute hosts with 128,000 slots, processing over 400 million tasks per day in production at major financial institutions with scheduling latency of 1 millisecond [10], [11]. Production deployments at individual financial institutions exceed hundreds of thousands of cores across federated Symphony clusters governed by one consumer hierarchy. Top banks worldwide run Symphony in production. Through LSF, the same ontology managed 27,648 GPUs across 4,608 nodes at Oak Ridge Summit [17], [18]. GPFS is NVIDIA-certified for DGX SuperPOD at up to 40 GBps read throughput per node [22]. Symphony's compute ontology (resource schema, consumer hierarchy, SOAM lifecycle) is defined independently of the number of resources it governs and scales without architectural modification.

The per-cluster ceiling of 128,000 slots is not Symphony's ceiling. Overflow federates multiple clusters under one governance domain with policy-based execution, configurable thresholds and session-based and task-based routing granularity. HostFactory provisions cloud GPU instances (AWS, Azure, GCP, IBM Cloud) into the same federated domain. Symphony's workload management scales horizontally by adding clusters, clouds, geographic sites and accelerator platforms, each extending the domain without architectural modification. Run:ai and Kueue inherit the Kubernetes cluster size ceiling (typically 5,000–15,000 nodes depending on Kubernetes version and etcd configuration). Run:ai's centralized management plane provides unified policy across clusters but scheduling remains confined to each cluster independently with no cross-cluster workload routing. Kueue's MultiKueue provides static multi-cluster dispatch at alpha maturity with significant functional limitations including no ResourceClaim synchronization and no quota consistency across clusters.

VIII. Discussion

A. The NVIDIA Vertical Integration

NVIDIA's acquisitions of Run:ai (completed January 2025) and SchedMD (announced December 2025) consolidate the scheduling layer under the GPU vendor [3], [23], [24]. The consolidation creates a vertically integrated stack in which NVIDIA controls the GPU hardware (DGX, HGX), the cluster management software (Base Command Manager), the Kubernetes-based GPU orchestrator (Run:ai) and the HPC batch scheduler (Slurm). The compute ontology proposed in the present paper provides a vendor-independent alternative grounded in four decades of production deployment rather than recent acquisitions.

The vendor independence argument requires symmetry. Symphony is an IBM proprietary product and dependence on IBM's platform introduces its own form of vendor risk. The distinction is architectural rather than commercial. Symphony's compute ontology is not tied to a specific GPU vendor's hardware. An enterprise running Symphony can deploy on NVIDIA, AMD, or Intel GPU/HPUs without compute ontology replacement. Run:ai supports NVIDIA GPUs only. As AMD and Intel gain inference market share through MI300X and Gaudi accelerators, the NVIDIA-only constraint becomes increasingly significant.

TABLE V
THREE-WAY CAPABILITY COMPARISON

Capability	Run:ai (v2.24)	Kueue (v0.17)	Symphony Compute Ontology
Resource type system	K8s resources + fractional GPU memory + MIG	K8s resources + Resource Flavors (node labels)	Site-defined typed schema with semantic direction
Real-time perception	Prometheus, 15–30s, dashboards only	None (admission controller)	ELIM, sub-second, feeds scheduling directly
Governance depth	4 levels, no nesting, production	Arbitrary depth, alpha API, known bugs	Unlimited depth, production, organizational digital twin
Lending/borrowing	Over-quota weights, reclaim on demand	Per-flavor limits, no automatic flow	Automatic flow through tree, per-consumer per-RG, 1s reclaim
Rebalancing	Scheduling-cycle-driven	Event-driven, no periodic sweep	Periodic (1s), redistributes running workloads
Preemption	Pod-level, grace period	Workload-level, triggered by new pending only	Task-level, proactive 4-level escalation
Request routing	Pod placement (Knative for request queuing)	None	Per-request, 38-metric scoring, ~10 ms
Semantic routing	None	None	resReq expressions + KNN, 91.1% accuracy
KV cache awareness	NIM metrics available, not used for scheduling	None	ELIM-reported, feeds routing and cache transfer
Service lifecycle	Pod lifecycle (pending/running/failed)	Admission gate (suspended/unsuspended)	SOAM 7-phase, per-phase failure policies
Session management	None	None	Multi-SSM failover, session continuity
Cross-cluster scheduling	None (centralized management only)	MultiKueue (alpha, static, limited)	Overflow: policy/threshold/session/task-based
Cloud bursting	None	None	HostFactory into same scheduling domain
Hardware platforms	NVIDIA (full), AMD (limited), Intel (none)	Hardware-agnostic via node labels	Any via ELIM: NVIDIA, AMD, Intel, neuromorphic, quantum
Scaling ceiling	Single K8s cluster	Single K8s cluster	Unbounded (clusters + clouds + sites + platforms)
Coordination substrate	K8s storage pass-through	None	GPFS: RDMA, xattrs, ILM, GPUDirect Storage
Data processing	None	None	Spark-on-EGO (Conductor), Jupyter, conda, Dask, DLI
GPU API access	Public NVML/DCGM/CUDA (no proprietary access)	N/A	Same public NVML/DCGM/CUDA + GPUDirect Storage on GPFS

B. The Three-Product Problem

NVIDIA's inference stack for enterprise deployments comprises Run:ai for container placement, Dynamo for inference routing and llm-d for distributed inference coordination. The three products exist because each addresses a gap the others leave. Run:ai places containers but does not route requests. Dynamo routes requests but does not manage container lifecycle or GPU governance. llm-d coordinates distributed inference but requires Kubernetes infrastructure for deployment.

Symphony's compute ontology collapses the three-product stack into one framework. ELIM provides the metric visibility Dynamo needs for routing decisions. SOAM provides the service lifecycle llm-d needs for distributed inference coordination. Consumer hierarchies provide the governance Run:ai provides through namespace quotas. The consolidation is not feature engineering but ontological completeness, since resource typing, governance, routing, service lifecycle and cross-substrate federation are aspects of one framework rather than separate products.

Prior work demonstrated that 16 of llm-d's routing algorithms and 15 additional capabilities reimplement natively on Symphony SOAM with GPFS-based state sharing [14], [15]. The present paper extends the integration by showing that Run:ai's GPU scheduling capabilities are also replicable within the same framework.

C. The Managed Kubernetes GPU Density Constraint

The demonstrations reveal a structural mismatch between managed Kubernetes GPU offerings and large model inference requirements. IBM Cloud's ROKS GPU worker profiles limit GPU density to 1–2 A100 GPUs per worker node. The limitation is an IBM Cloud implementation constraint rather than an OpenShift architectural constraint; OpenShift itself runs on 8-GPU DGX nodes at enterprises worldwide and NVIDIA's own DGX SuperPOD reference architecture supports OpenShift on 8×H100 nodes. Similar GPU density limitations exist across managed Kubernetes offerings from other cloud providers, where GPU instance profiles are designed for general-purpose cloud workloads rather than for GPU-dense AI inference.

The mismatch has concrete operational consequences. Granite 34B at fp16 precision requires approximately 68 GB of GPU memory, exceeding the capacity of a single A100 (80 GB) when accounting for KV cache and runtime overhead and requiring multi-GPU tensor parallelism for acceptable latency. A managed ROKS worker node with 1–2 A100 GPUs cannot serve Granite 34B. An enterprise deploying multi-tier inference spanning 2B to 34B parameter models on managed OpenShift must either limit its model range to what

the GPU worker profiles support or deploy additional GPU infrastructure outside the managed Kubernetes cluster.

Run:ai, confined to the Kubernetes cluster boundary, cannot route to GPU resources outside the cluster. The enterprise is forced to choose between limiting model range (serving only models that fit managed K8s GPU profiles) or operating a separate inference stack for large models (fragmenting workload management). Symphony's compute ontology resolves the dilemma by spanning substrates. Lightweight models serve on managed OpenShift with its operational maturity and security certification. Large models serve on bare-metal hosts with their GPU density. Mid-range models serve on self-managed OpenShift with GPU passthrough on commodity hardware. All three substrates participate in one compute ontology under one consumer hierarchy with one set of ELIM metrics. The cross-substrate capability is not a theoretical advantage but an operational necessity arising from the real GPU density constraints of managed cloud Kubernetes.

D. Relationship to Prior Work

The compute ontology extends the architectural arguments established in three companion papers. The multi-ontology AI factory paper [14] established LSF and Symphony as complementary compute ontologies for training and inference, with GPFS as coordination substrate. The quantum-centric heterogeneous orchestration paper [15] established ELIM's accommodation of resources differing in kind across QPU, NPU, GPU, CPU and mainframe tiers. The Palantir sovereign AI OS paper [8], [28] established Symphony as a compute ontology parallel to Foundry's data ontology. The present paper extends that insight into the commercially critical domain of OpenShift and NVIDIA AI factory infrastructure, where Run:ai and NVIDIA's vertical integration present both competitive risk and architectural opportunity.

E. Limitations

The Run:ai comparison uses the open-source KAI Scheduler and publicly documented Run:ai v2.24 capabilities. The commercial Run:ai platform may include undocumented capabilities not captured in this analysis.

Performance figures cited from prior work (approximately 10 ms routing decisions, 340 ms KV cache transfer, 91.1% semantic routing accuracy [14], [15]) were measured on bare-metal Symphony deployments. The cross-substrate topology described in the present paper (ELIM agents on OpenShift, metrics traversing Kubernetes networking, GPFS accessed through CSI drivers) may introduce additional latency. Part II will revalidate these figures in the OpenShift integration context across three compute substrates with different GPU hardware (A100, RTX 3090).

The consumer hierarchy governance claims at institutional scale rest on Symphony's demonstrated production deployments at major financial institutions rather than on institutional-scale testing within the demonstration environment [10], [11].

Deploying Symphony alongside OpenShift introduces a second resource management framework. The operational footprint of Symphony (EGO resource broker, LIM/ELIM metric infrastructure, SOAM service managers) is comparable to the operational footprint of an enterprise Kubernetes deployment (OLM, ArgoCD, Tekton, Prometheus, Grafana, Istio, cert-manager, external-dns, storage operators and the dozens of CRDs required before a single workload runs). Neither platform is operationally simple at enterprise scale. The three engagement modes (Section V.B) accommodate different levels of integration and Mode 2 (Ontology Governance) requires zero modification to the OpenShift cluster.

IX. Future Work

Integration of the compute ontology with Palantir Foundry through the Ontology MCP protocol would enable Foundry ontology operations to participate as typed SOAM service invocations with ELIM-reported ontology health metrics, closing the loop between data intelligence and compute intelligence.

Formal benchmarking of the compute ontology against Run:ai on matched DGX SuperPOD hardware at institutional scale would provide the quantitative comparison the architectural analysis motivates.

Extension of the ELIM perception model to capture Kubernetes-native metrics (pod phase, restart count, resource limits) as typed resources within the compute ontology would provide a complete operational view spanning both the container and workload layers.

Neuromorphic predictive resource management, in which AKD1000 spiking neural networks classify infrastructure telemetry at sub-millisecond latency, would detect demand regime changes before they manifest as scheduling failures, enabling the compute ontology to act proactively rather than reactively.

X. Conclusion

Kubernetes is the enterprise AI platform. The hardware is deployed. The networking is operational. The container infrastructure is mature. The compute intelligence layer remains the last architectural gap and the gap exists because Kubernetes is a container placement engine, not a compute ontology.

The present paper, the first of two parts, addresses the gap by establishing IBM Spectrum Symphony as the compute ontology extending insight into OpenShift and NVIDIA AI factory infrastructure, with comprehensive comparative analysis. The compute ontology provides typed resource metrics with semantic direction through ELIM, hierarchical consumer governance with sub-second rebalancing, SOAM service lifecycle management with per-phase failure policies, cross-substrate routing spanning OpenShift clusters, bare-metal GPU hosts, cloud burst instances and heterogeneous accelerators under one scheduling domain and GPU-accelerated data processing through Spectrum Conductor on the same EGO resource broker.

The comparative analysis establishes three findings. Kueue, Kubernetes' strongest governance extension, operates at alpha

API maturity with known defects, no periodic rebalancing of running workloads and zero capability in five of six ontological dimensions. Run:ai exclusively uses public GPU APIs with no proprietary hardware access and every Run:ai capability is replicable within the compute ontology with greater depth. NVIDIA's own selection of GPFS for GPU-native analytics at GTC 2026 validates the coordination substrate the compute ontology uses.

The compute ontology is the structural parallel to Palantir Foundry's data ontology. Foundry models the data domain. Symphony models the compute domain. Kubernetes manages the infrastructure. The three layers serve different categorical needs. An enterprise running all three has typed, governed, semantically rich models of both its data domain and its compute domain, operating on an enterprise platform proven at institutional scale.

The implementation lineage enabling the compute ontology spans 39 years, from Zhou's 1987 load index through Platform Computing, Platform Symphony and IBM Spectrum Symphony. The same principle governing the architecture, measuring each resource according to its own design and managing workloads on the basis of those measurements within a unified domain, has scaled from VAX workstations to AI factories. The principle works because it is correct. The compute domain needs an ontology. Symphony provides it. OpenShift provides the infrastructure. NVIDIA provides the GPUs. A companion paper (Part II) will provide empirical validation across three compute substrates.

References

[1] NVIDIA, "AI Factory White Paper: Ecosystem Architecture," 2026. Available: https://docs.nvidia.com/ai-enterprise/planning-resource/ai-factory-white-paper/latest/ecosystem-architecture.html

[2] NVIDIA, "Run:ai GPU Orchestration Platform," integrated into NVIDIA AI Enterprise, 2025. Available: https://www.run.ai/

[3] TechCrunch, "NVIDIA Completes Acquisition of Run:ai," Dec. 30, 2024. Available: https://techcrunch.com/2024/12/30/nvidia-completes-acquisition-of-ai-infrastructure-startup-runai/

[4] NVIDIA, "Dynamo: A Framework for Efficient and Scalable AI Inference Serving," 2026. Available: https://developer.nvidia.com/dynamo

[5] Red Hat Developer, "llm-d: Kubernetes-native distributed inferencing," May 2025. Available: https://developers.redhat.com/articles/2025/05/20/llm-d-kubernetes-native-distributed-inferencing

[6] S. Zhou, "Performance Studies of Dynamic Load Balancing in Distributed Systems," Ph.D. dissertation, Dept. of Electrical Engineering and Computer Sciences, Univ. of California, Berkeley, Tech. Rep. UCB/CSD-87-376, Oct. 1987.

[7] Palantir Technologies, "Foundry Platform Overview: Architecture." Available: https://www.palantir.com/docs/foundry/platform-overview/architecture

[8] K. D. Johnson, "Extending the Sovereign AI OS: Symphony as Compute Ontology for Palantir Foundry and NVIDIA," Technical Paper, March 2026.

[9] S. Zhou, X. Zheng, J. Wang and P. Delisle, "Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," *Software: Practice and Experience*, vol. 23, no. 12, pp. 1305–1336, Dec. 1993.

[10] IBM, *IBM Spectrum Symphony documentation*, including Service-Oriented Architecture Middleware (SOAM), External Load Information Manager (ELIM), HostFactory, Overflow and Consumer Hierarchies. Available: https://www.ibm.com/docs/en/spectrum-symphony

[11] D. Quintero *et al.*, *IBM Platform Computing Solutions*, IBM Redbook SG24-8073, Dec. 2012. Available: https://www.redbooks.ibm.com/abstracts/sg248073.html

[12] insideHPC, "Songnian Zhou: Why Combine Platform and IBM?," Oct. 2011. Available: https://insidehpc.com/2011/10/songnian-zhou-why-combine-platform-and-ibm/

[13] HPCwire, "From Clusters to Clouds: An Interview with Platform CEO Songnian Zhou," Jun. 15, 2010. Available: https://www.hpcwire.com/2010/06/15/from_clusters_to_clouds_an_interview_with_platform_ceo_songnian_zhou/

[14] K. D. Johnson, "Solving the One and the Many with LSF, Symphony, GPFS and RHEL AI: A Dynamic Compute Platform for NVIDIA AI Factories," Technical Paper, March 2026.

[15] K. D. Johnson, "High Performance Quantum-Centric Supercomputing: A Working Implementation of Heterogeneous Orchestration across QPU, NPU, GPU, CPU and Other Tiers," Technical Paper, March 2026.

[16] NVIDIA, "KAI Scheduler," GitHub repository. Available: https://github.com/NVIDIA/KAI-Scheduler. See Issues #848, #1217, #423.

[17] NVIDIA, "IBM Spectrum LSF," NVIDIA Developer documentation. Available: https://developer.nvidia.com/ibm-spectrum-lsf. See also: B. McMillan, "DynaMIG Management of NVIDIA DGX A100 with IBM Spectrum LSF," IBM Community Blog, Jan. 2021.

[18] Oak Ridge National Laboratory, "Summit User Guide," 2018–2024. Available: https://docs.olcf.ornl.gov/systems/summit_user_guide.html. See also: A. Bland *et al.*, "Scaling the Summit," OSTI Technical Report, 2018.

[19] L. Chen, M. Zaharia and J. Zou, "FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance," arXiv:2305.05176, 2023.

[20] I. Ong *et al.*, "RouteLLM: Learning to Route LLMs with Preference Data," arXiv:2406.18665, 2024.

[21] IBM, *IBM Storage Scale (GPFS) documentation*. Available: https://www.ibm.com/docs/en/storage-scale

[22] IBM, *IBM Storage Scale System 6000 with NVIDIA DGX SuperPOD*, IBM Redbook REDP-5746. Available: https://www.redbooks.ibm.com/abstracts/redp5746.html

[23] The Next Platform, "NVIDIA Nearly Completes Its Control Freakery with Slurm Acquisition," Dec. 18, 2025. Available: https://www.nextplatform.com/2025/12/18/nvidia-nearly-completes-its-control-freakery-with-slurm-acquisition/

[24] A. B. Yoo, M. A. Jette and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing* (LNCS 2862), Springer, 2003, pp. 44–60.

[25] Red Hat, "Red Hat Enterprise Linux AI (RHEL AI)." Available: https://www.redhat.com/en/products/ai/enterprise-linux-ai

[26] W. Kwon *et al.*, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proc. SOSP '23*, 2023.

[27] S. Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1327–1341, Sep. 1988.

[28] Palantir Technologies and NVIDIA, "Palantir and NVIDIA Team to Deliver Sovereign AI Operating System Reference Architecture," Mar. 2026. Available: https://investors.palantir.com/news-details/2026/

[29] IBM, "IBM Cloud for Financial Services." Available: https://www.ibm.com/cloud/financial-services

[30] Kubernetes SIG Scheduling, "Kueue: Kubernetes-native Job Queueing." Available: https://kueue.sigs.k8s.io/. See also: Cohort CRD (v1alpha1), GitHub Issue #3948.

[31] IBM, *IBM Spectrum Conductor documentation*, including Spark-on-EGO, Jupyter notebook management, Anaconda/conda environments and Deep Learning Impact. Available: https://www.ibm.com/docs/en/spectrum-conductor

[32] NVIDIA, "RAPIDS cuDF: GPU DataFrame Library." Available: https://rapids.ai/cudf. See also: kvikIO (GPUDirect Storage I/O library), https://github.com/rapidsai/kvikio

[33] IBM and NVIDIA, "IBM and NVIDIA Announce Expanded Collaboration at GTC 2026 to Advance AI for the Enterprise," IBM Newsroom, Mar. 16, 2026. Available: https://newsroom.ibm.com/2026-03-16-ibm-and-nvidia-announce-expanded-collaboration-at-gtc-2026

[34] NVIDIA, "NVIDIA AI Enterprise Licensing Guide." Available: https://docs.nvidia.com/ai-enterprise/planning-resource/licensing-guide/latest/licensing.html