The Semantics of a Resource Request: Moving Beyond Cores and Gigabytes

Shreya Gupta

University of Southern California, USA

Abstract

This article challenges traditional resource request models in cluster computing that rely on abstract CPU core and memory specifications, particularly for modern AI/ML workloads. Current coarse-grained abstractions mask essential hardware characteristics and workload requirements, causing performance unpredictability, resource inefficiency, and service level violations. By examining the limitations of current models, the paper exposes performance non-fungibility issues and contention on unmanaged resources like memory bandwidth and cache hierarchies. A new multi-dimensional taxonomy of resource semantics encompasses hardware attributes, workload behaviors, and operational policies. Through analysis of schedulers from HPC, hyperscale, and open-source domains, the article shows how richer semantics are gradually being adopted. The research addresses opportunities and challenges of this semantic shift and explores future directions, including standardized Resource Description Languages and the application of Large Language Models for intelligent scheduling. The article advocates for semantically-aware schedulers capable of intelligent, topology-aware resource matching to enhance performance and efficiency in large-scale systems.

Keywords: Resource semantics, cluster scheduling, hardware-aware matchmaking, workload characterization, multi-dimensional taxonomy

1. Introduction

1.1 The Evolution of Resource Abstraction

The history of large-scale computing parallels the evolution of resource abstraction.[1] From mainframe time-sharing in the 1960s, the central challenge has been efficiently multiplexing expensive hardware among competing users and applications.[1][2] The progression through distributed systems, cluster computing, and grid computing represents continuous efforts to manage increasingly complex machine collections.[2] Cloud computing's emergence in the early 2000s marked a pivotal moment, driven by technologies like virtualization, offering Infrastructure as a Service (IaaS).[1]

Central to this paradigm was a straightforward but effective abstraction: the virtual machine (VM), specified by virtual CPU cores (vCPUs) and gigabytes of memory.[2] This "cores and gigabytes" model served as a necessary simplification, allowing cloud providers to create fungible resource pools from heterogeneous hardware, enabling on-demand, self-service provisioning that defined cloud computing.[2] For transaction-based, stateless web services, and enterprise applications—the dominant workloads of that era—this abstraction sufficed.[3] It successfully concealed resource management complexities and failure handling, letting developers focus on application logic rather than infrastructure details.[4] The model aimed to make diverse hardware appear homogeneous, succeeding remarkably and paving the way for cloud-native computing growth.[4][5]

1.2 The Rise of Semantically Demanding Workloads

Recent years have witnessed the emergence of Artificial Intelligence and Machine Learning (AI/ML) workloads dominating large-scale computing.[6][5] From training massive foundation models to deploying latency-sensitive inference services, these applications exhibit fundamentally different characteristics from their predecessors.[5] Computationally intensive, data-hungry, and often distributed, their performance characteristics are tightly coupled with underlying hardware specifics.[6]

For these semantically demanding workloads, the "cores and gigabytes" abstraction proves not merely an oversimplification but a significant liability.[6][7] A request for 8 cores, 32GB RAM conveys nothing about processor instruction sets, memory hierarchy, Non-Uniform Memory Access (NUMA) topology, or network interconnect bandwidth.[7] Yet, for distributed training jobs, these factors determine performance. Placing communication tasks on nodes separated by high-latency network links can cripple performance, a detail invisible to schedulers that only count vCPUs.[6] Similarly, co-locating memory-bandwidth-intensive data preprocessing with cache-sensitive model training can cause unpredictable performance degradation due to contention on shared, unmanaged resources.[6] The abstraction, designed to hide details, now conceals information necessary for efficient execution, creating a severe "impedance mismatch" between application needs and scheduler understanding of available resources.[7][8]

1.3 Thesis and Contributions

The success of the vCPU/RAM abstraction has created a form of technical debt in system design.[9] It allowed deferring confrontation with growing hardware complexity for over a decade.[9][10] The rise of AI/ML now calls that debt due, forcing the systems community to address this oversimplified model's limitations.[10] The very design choice enabling the cloud's initial growth now becomes a primary bottleneck for its most valuable and fastest-growing workloads.[11][12]

This paper posits that the next frontier in cluster management requires a paradigm shift: moving beyond coarse-grained resource counting to a model based on rich, multi-faceted semantics.[12] Schedulers must evolve from simple resource packers to intelligent, hardware-aware matchmakers.[13] The paper makes these contributions:

- A critical analysis of vCPU/RAM abstraction failure modes, demonstrating inadequacy for modern, performance-sensitive workloads.[14][15]
- A structured taxonomy for a richer resource description framework, providing vocabulary for expressing hardware capabilities, workload requirements, and policy objectives.[15]
- A comparative analysis of how leading schedulers from High-Performance Computing (HPC), hyperscale, and open-source domains address this semantic gap with partial solutions.[16]
- An exploration of future challenges and opportunities presented by this semantic shift, including standardization needs and AI's potential role in next-generation schedulers.[17][18]

1.4 Roadmap

The paper continues as follows. Section 2 deconstructs current resource model limitations in detail.[19] Section 3 proposes a taxonomy of rich resource semantics addressing these limitations.[19] Section 4 reviews influential scheduling systems through this taxonomy's lens.[20] Section 5 discusses broader opportunities and challenges of adopting a semantic approach.[20] Section 6 explores future research directions, and Section 7 concludes the paper.[21]

2. The Fragility of Abstraction: Limitations of Current Models

The "cores and gigabytes" model assumes resources are fungible—that CPU cores are interchangeable, and RAM gigabytes represent simple, quantifiable units.[22] For modern hardware and workloads, this assumption proves demonstrably false.[22][23] This section systematically deconstructs the abstraction's failure modes, showing how it leads to performance unpredictability, inefficiency, and unfairness.[23]

2.1 Performance Non-Fungibility: Not All Cores are Created Equal

"vCPU" itself serves as an abstraction, typically representing a time slice on a physical CPU core managed by a hypervisor or operating system scheduler.[8] This simple abstraction masks the underlying micro-architectural complexity, having a first-order impact on application performance.[8][9]

2.1.1 Instruction Set Architecture (ISA) Impact

Modern CPUs feature complex instruction sets with specialized extensions designed for specific computation types.[9] Prominent examples include Single Instruction, Multiple Data (SIMD) extensions like Intel's Advanced Vector Extensions (AVX), particularly AVX2 and AVX-512.[9] These instructions can process twice (AVX2) or four times (AVX-512) the data elements per instruction compared to legacy instructions, offering significant performance gains for scientific computing and deep learning.[9]

This performance comes with costs.[10] Executing "heavy" 256-bit or 512-bit AVX instructions consumes significantly more power and generates more heat than standard instructions.[10] To stay within thermal design power (TDP) envelopes, CPUs automatically reduce operating frequency when executing such code.[10] An Intel Xeon Gold 6130

CPU might run non-AVX code at 2.8 GHz but downclock to 2.4 GHz for AVX2 code and as low as 1.9 GHz for heavy AVX-512 code.[10] Crucially, this frequency reduction often applies core-wide or even package-wide, affecting all co-located tasks, not just the one executing AVX instructions.[10]

This creates profound scheduling problems.[11] A scheduler allocating equal CPU time to an AVX-512 task and a co-located non-AVX task fails to allocate equal performance.[11] The non-AVX task (the "victim") receives its time slice at significantly reduced clock speed, suffering performance penalties up to 25% through no fault of its own.[11]

Business Impact: For latency-sensitive microservices, this 25% slowdown can mean the difference between meeting and missing SLAs. In e-commerce environments, where every 100ms of latency reduces conversion rates by 7%, this hidden performance penalty directly translates to lost revenue, potentially millions of dollars annually for large operations. For financial trading systems, where microseconds matter, these unplanned slowdowns can cost firms $100,000 or more per millisecond of added latency in competitive markets.

A scheduler blind to ISA semantics cannot reason about this effect, leading to unfairness and SLO violations.[11] It might unknowingly place latency-sensitive web servers alongside scientific computing jobs, causing web server performance to degrade unpredictably.[11][12]

2.1.2 NUMA and Memory Locality

Modern multi-socket servers almost universally employ Non-Uniform Memory Access (NUMA) architecture.[6] In NUMA systems, each CPU socket has local memory banks.[6] While any CPU can access any system memory, accessing local memory proves significantly faster and offers higher bandwidth than accessing "remote" memory attached to another socket, which requires traversing slower interconnects like Intel's UPI or AMD's Infinity Fabric.[6]

For memory-intensive AI/ML workloads constantly moving large data volumes between CPU memory and accelerators, NUMA locality becomes critical.[6] Placing data preprocessing processes on CPUs in one NUMA node while data resides in another node's memory introduces severe performance bottlenecks, with studies showing degradation up to 700%.[6]

Business Impact: This 7x slowdown transforms what should be a 2-hour model training job into a 14-hour ordeal, potentially:

- Delaying time-to-market for AI products by days or weeks
- Multiplying cloud computing costs by 7x (turning a $1,000 training run into a $7,000 expense)
- Reducing data scientists productivity as they wait for results
- Forcing companies to purchase 7x the computing resources needed with proper scheduling

This effect becomes particularly pronounced during deep learning training data loading phases, where CPU worker processes feed data batches to GPUs.[6] If worker processes, memory buffers, and target GPUs aren't all located within the same NUMA domain, resulting cross-node traffic can starve GPUs and negate parallel processing benefits.[6]

NUMA-unaware schedulers treat all memory as a single pool and all cores as equivalent.[12] They might schedule multi-threaded applications across multiple NUMA nodes or place VM vCPUs on one socket and memory on another, inadvertently creating performance-killing remote memory access patterns.[12] This demonstrates that cores are non-fungible, and the relationship between cores and memory represents a critical, yet hidden, semantic relationship.[12][13]

2.2 The Noisy Neighbor Problem 2.0: Contention on Unmanaged Resources

The classic "noisy neighbor" problem in multi-tenant environments describes contention for explicitly managed resources like CPU time, network I/O, and disk I/O.[16] However, dense, multi-core architectures have created a more insidious version: contention on shared, unmanaged micro-architectural resources.[16][17] Even with perfect CPU and memory isolation at container levels, co-located tasks can interfere through these hidden side channels.[17]

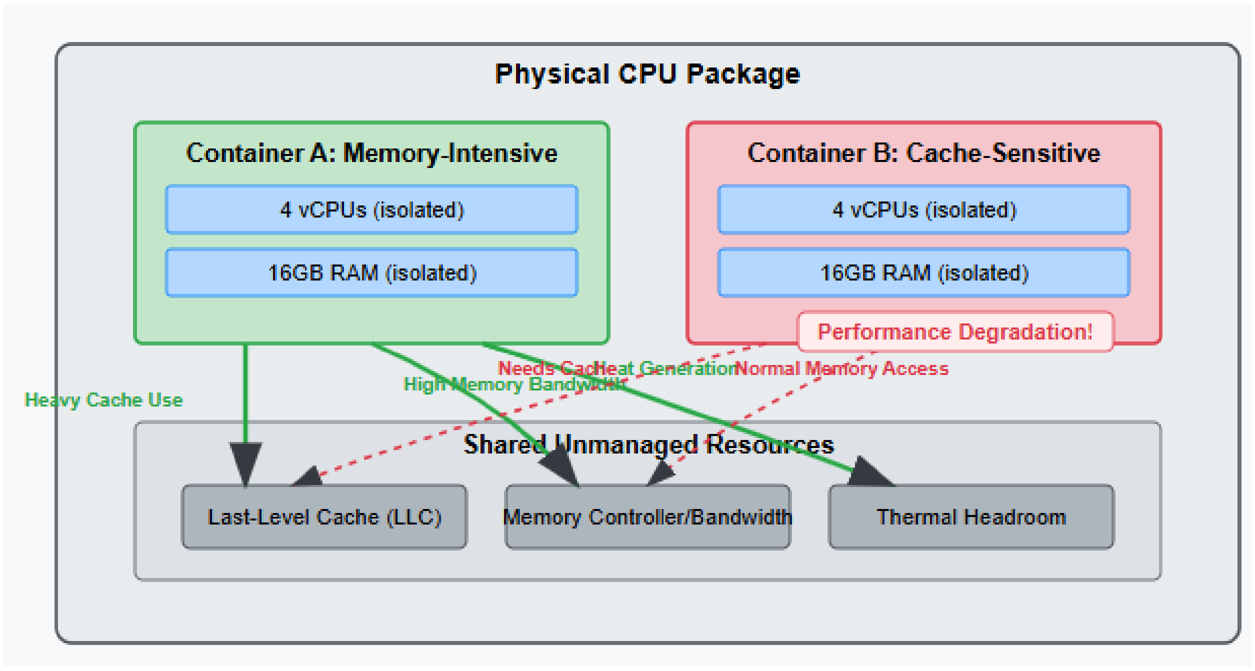

Figure 1: Visualization of the "Noisy Neighbor 2.0" problem. Two containerized applications with perfect CPU and memory quota isolation still experience performance interference through shared, unmanaged resources like last-level cache, memory bandwidth, and thermal headroom. The memory-intensive Application A (left) floods the shared memory bus and evicts Application B's cache lines, causing Application B (right) to experience unpredictable performance degradation despite having its guaranteed CPU resources.

2.2.1 Memory Bandwidth Contention

The interconnect between processors and main memory has become a primary performance bottleneck on modern servers.[7] While schedulers can limit container RAM consumption, they typically cannot control memory access rates.[7] When multiple memory-intensive applications co-locate on the same server (or socket), they compete for finite memory bandwidth.[7] This contention leads to super-linear performance degradation, where the slowdown experienced by each application exceeds what simple resource sharing would predict.[7]

Business Impact: In data analytics workloads, memory bandwidth contention can reduce throughput by up to 35%, meaning:

- A daily ETL pipeline that normally completes in 4 hours now takes 5.4 hours
- Customer-facing dashboards experience unpredictable latency spikes of 2-3 seconds
- Real-time analytics systems miss their processing windows, leading to stale insights for business decisions
- For a 1,000-node cluster, this translates to approximately $1.2M in wasted compute resources annually

Memory access scheduling, which reorders memory operations to optimize DRAM bank and row utilization, can improve bandwidth by over 90% in some cases, highlighting performance sensitivity to access patterns.[15] Memory bandwidth-aware schedulers can co-schedule jobs to avoid saturating memory buses, for example, by pairing compute-bound tasks with memory-bound tasks, improving overall system throughput.[16] Schedulers blind to this semantic dimension make placement decisions that inadvertently create memory bandwidth hotspots, leading to unpredictable performance.[16][17]

2.2.2 Cache Contention

Modern CPUs rely on multi-level cache hierarchies to bridge performance gaps between processors and main memory.[17] The last-level cache (LLC), typically shared among all socket cores, represents a critical shared resource.[17] When multiple applications co-locate, they compete for LLC space.[17] "Thrashing" applications with large memory footprints and poor locality repeatedly evict useful data of co-located cache-sensitive applications, forcing them to fetch data from much slower main memory.[17] This interference severely degrades performance.[18]

Business Impact: For cache-sensitive applications like database systems, LLC contention can reduce query performance by 60-80%:

- A financial application that normally processes 10,000 transactions per second might handle only 2,000-4,000 TPS
- User-facing response times increase from milliseconds to seconds
- In e-commerce, this translates to shopping cart abandonment rates increasing by up to 30%
- For mission-critical applications, businesses often over-provision by 2-3x to compensate for this unpredictability, wasting millions in infrastructure costs

Research into cache-aware scheduling demonstrates the importance of managing this resource.[18] Techniques like cache partitioning, where LLCs are explicitly divided among applications, can isolate tasks, improving predictability and performance.[18] Schedulers' understanding of application cache requirements and sensitivity can make more intelligent co-location decisions, preventing destructive interference.[20]

2.2.3 Interconnect Contention (GPUs)

For distributed AI/ML workloads running on multiple GPUs, the interconnect topology and bandwidth between accelerators become paramount.[21] High-speed, direct GPU-to-GPU interconnects like NVIDIA's NVLink offer significantly higher bandwidth (up to 40 GB/s per link) than general-purpose PCIe buses.[21] Performance of communication-intensive operations, such as Allreduce collectives common in data-parallel training, depends directly on communication paths between participating GPUs.[22]

Topology-unaware schedulers might place heavily communicating tasks on GPUs in different sockets, forcing traffic over slower system buses and PCIe links instead of direct NVLink bridges.[21] This can result in performance speedup differences up to 1.30x simply based on placement within single, NVLink-enabled machines.[21]

Business Impact: For large foundation model training, suboptimal GPU communication pathways can extend training time by 20-30%.

- A model that should train in 10 days takes 13 days
- For models costing $2-3M in compute resources to train, this represents $400,000-$900,000 in wasted resources
- Time-to-market delays can cost a competitive advantage in fast-moving AI fields
- For inference services, these inefficiencies can require 20-30% more GPU hardware to maintain the same throughput, significantly increasing operational costs

This problem becomes more pronounced in multi-node clusters, where inter-node network performance becomes another critical factor.[22] Truly semantic schedulers must treat interconnect topology not as implementation details but as first-class schedulable resources.[22][23]

2.3 Mismatch with AI/ML Execution Models

Beyond hardware-level mismatches, the "cores and gigabytes" model fundamentally misaligns with modern AI/ML workload execution models.[6] Cluster schedulers like Kubernetes were designed around concepts of single, independent, often stateless containers (pods).[6] This abstraction maps poorly to complex, stateful, multi-component AI jobs.[6][5]

2.3.1 The Job, Not the Task

The fundamental work unit for data scientists or ML engineers is the job—a distributed training run, hyperparameter tuning sweep, or multi-stage data processing pipeline.[6] Individual containers or pods merely represent implementation details.[6] However, default Kubernetes schedulers operate at pod level, without a holistic understanding of the jobs they belong to.[6] This creates several critical issues. When jobs consisting of many pods are submitted, schedulers see floods of independent scheduling requests.[6] This can overwhelm schedulers and underlying etcd data stores, degrading overall cluster performance.[6] More importantly, it prevents schedulers from making globally optimal decisions for jobs as whole.[6][23]

Business Impact: The mismatch between job-level and pod-level scheduling creates severe operational inefficiencies:

- Large ML experiments can take 2-3x longer to complete due to suboptimal pod placement
- Platform engineers spend 15-20% of their time troubleshooting and manually optimizing ML workloads
- Organizations maintain 30-40% excess capacity to compensate for scheduling inefficiencies
- For a mid-sized ML platform with 1,000 GPUs, this represents $2-3M annually in unnecessary hardware costs

2.3.2 Lack of Gang Scheduling

Many distributed AI frameworks using data parallelism require all constituent tasks (workers, parameter servers) to be scheduled concurrently to make progress.[5] If schedulers place only subsets of required pods, entire jobs stall, holding allocated resources while waiting for remaining pods to be scheduled.[5] This all-or-nothing requirement is known as gang scheduling.[5] Default Kubernetes schedulers lack native gang scheduling concepts, placing pods one by one as resources become available.[5] This leads to deadlocks and significant resource wastage, problems so common they've spurred the development of specialized, job-aware schedulers like Volcano and YuniKorn layered atop Kubernetes.[6]

Business Impact: Lack of gang scheduling for distributed ML workloads results in:

- Resource utilization dropping by 25-35% in ML-focused clusters
- "Orphaned" GPU allocations where some workers are scheduled but others aren't, wasting expensive accelerator time
- For a cluster with 100 A100 GPUs (approximately $1.5M in hardware), this inefficiency costs $375,000-$525,000 in wasted capital expenditure
- Increased operational complexity as teams implement workarounds and custom solutions

2.3.3 Resource Fragmentation

AI workloads often require specific quantities of specialized resources, such as "4 NVIDIA A100 GPUs on a single node."[5] Clusters may have 8 free A100s, but if scattered as 2 GPUs on four different nodes, schedulers considering only total counts will fail to place jobs.[5] This resource fragmentation causes major underutilization of expensive accelerator hardware.[5] Semantically aware schedulers would understand resource request "shapes" (e.g., 4 GPUs, node-local) and could attempt to compact other workloads to create large enough contiguous resource blocks to satisfy requests.[5][22]

Business Impact: Resource fragmentation in AI clusters leads to:

- GPU utilization rates falling below 40% in heterogeneous clusters (compared to 70-80% possible with shape-aware scheduling)
- Organizations purchasing 2x the necessary hardware to ensure workload placement
- A Fortune 500 company with a 500-GPU AI platform might waste $5-7M annually due to fragmentation
- Data science teams facing unpredictable queue times, with projects delayed by days or weeks waiting for the exact resource shape

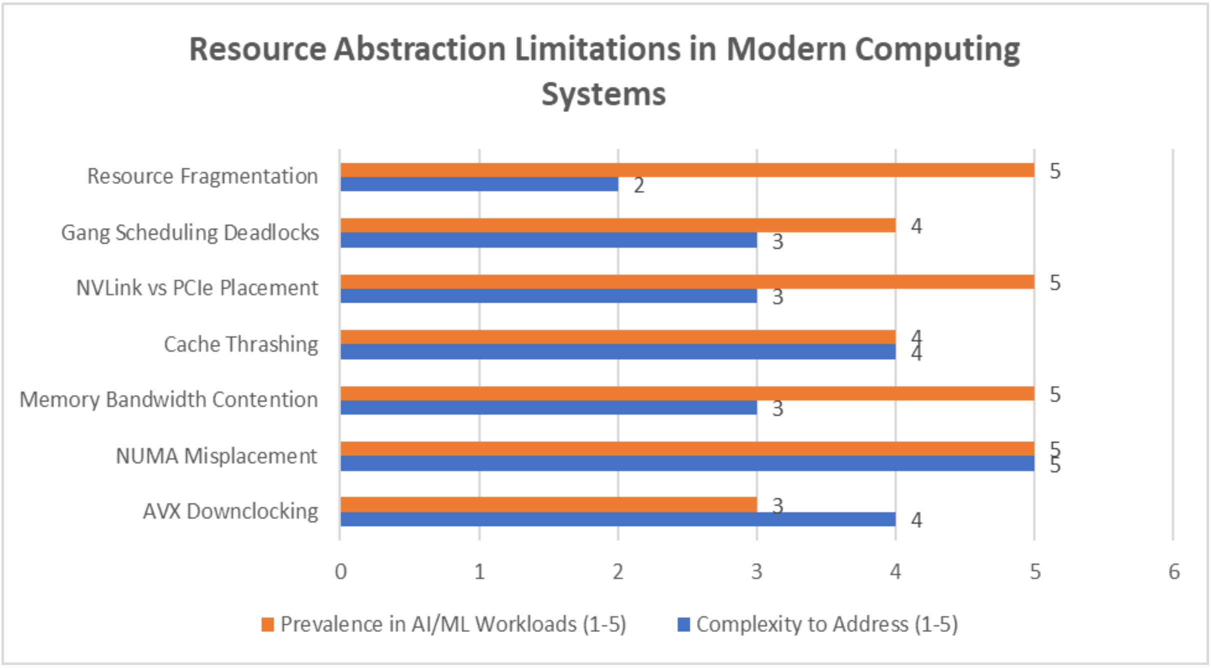

Figure 2: Performance Impact of Resource Scheduling Gaps in AI/ML Workloads

3. A Taxonomy of Rich Resource Semantics

Moving beyond "cores and gigabytes" model limitations requires new, more expressive vocabulary—allowing users and applications to communicate requirements and intent to schedulers with high fidelity.[5][19] This section proposes a structured taxonomy for these rich resource semantics, organizing them into four orthogonal dimensions: Hardware-Level (the "what"), Workload-Level (the "how"), Policy-Level (the "why"), and Security-Level (the "who and how safely").[19] This taxonomy serves as conceptual foundation for future standardized Resource Description Languages (RDLs).[19][46]

3.1 Hardware-Level Semantics (The "What")

This dimension describes intrinsic, physical capabilities of underlying resources.[19] It allows requests to target specific hardware features critical for performance, moving from abstract counts to concrete attributes.[19][20]

Compute: Specifies processing unit characteristics, including processor types and specific features.[20]

- cpu.architecture: x86_64, aarch64
- cpu.isa: [avx2, avx512]
- accelerator.type: {gpu: nvidia_a100, tpu: v4}
- accelerator.count: 4

Memory: Describes memory subsystem properties, including performance and topology.[20]

- memory.bandwidth: >200GB/s
- memory.hierarchy: {l3_cache: >24MB}
- numa.policy: strict_local (requesting all memory and CPUs for tasks be allocated from single NUMA nodes).

Storage: Defines attached storage types and performance characteristics.[20]

- disk.type: {local_nvme}
- disk.iops: >100k
- disk.capacity: >1TB

Network & Interconnect: Specifies data communication fabric capabilities, between nodes and components within nodes.[20][21]

- network.bandwidth: >100Gbps
- network.latency: low
- interconnect.topology: {type: nvlink, count: 8, fully_connected: true} (describing nodes with 8 fully-connected NVLink-enabled GPUs).

Example Hardware-Level RDL Specification

```
# Example YAML-like RDL specification for Hardware-Level semantics
hardware:
  compute:
    cpu:
      architecture: x86_64
      isa: [avx2, avx512]
      cores: 32
      threads_per_core: 2
    accelerator:
      type: nvidia_a100
      count: 4
      memory: 80GB
  memory:
    capacity: 512GB
    bandwidth: ">200GB/s"
    numa:
      policy: strict_local
      domains: 2
    hierarchy:
      l3_cache: ">24MB"
  storage:
    type: local_nvme
    capacity: 2TB
    iops: ">100k"
    throughput: ">5GB/s"
  network:
    bandwidth: ">100Gbps"
    latency: low
  interconnect:
```

```
  type: nvlink
  count: 8
  topology: fully_connected
  bandwidth_per_link: "40GB/s"
```

3.2 Workload-Level Semantics (The "How")

This dimension describes application behavior, structure, and resource sensitivities.[21] It provides schedulers crucial context about how workloads will use allocated resources, enabling more intelligent co-location and placement decisions.[21][22]

Execution Model: Defines job structure and dependencies.[21]

- job.type: batch, service, interactive
- job.structure: {gang_scheduled, size: 16} (jobs requiring 16 instances launched simultaneously).
- task.dependencies: dag (indicating directed acyclic graphs of tasks).

Resource Pressure: Characterizes workload sensitivity to contention on specific, often unmanaged, hardware resources.[17] This information proves vital for avoiding "Noisy Neighbor 2.0" problems.[17]

- sensitivity.memory_bandwidth: high
- sensitivity.l3_cache: high
- sensitivity.network_latency: critical
-

Communication Pattern: Describes data flow between tasks in distributed jobs, allowing topology-aware placement.[21][22]

- communication.pattern: {type: all_reduce, volume: high}
- data.locality_preference: strong (indicating jobs benefit significantly from data co-location with compute).

Example Workload-Level RDL Specification

```
# Example YAML-like RDL specification for Workload-Level semantics
workload:
  execution_model:
    job:
      type: batch
      structure:
        gang_scheduled: true
        size: 16
      lifetime:
        estimated_runtime: 4h
        max_runtime: 8h
    task:
      dependencies: dag
      checkpoint_interval: 30m
  resource_pressure:
    sensitivity:
      memory_bandwidth: high
      l3_cache: high
      network_latency: critical
      cpu_frequency: medium
      disk_iops: low
  communication_pattern:
    pattern:
      type: all_reduce
      volume: high
      frequency: high
    data:
      locality_preference: strong
      access_pattern: sequential
      working_set_size: 200GB
```

3.3 Policy-Level Semantics (The "Why")

This dimension captures high-level operational and business objectives governing scheduling decisions.[23] It allows operators to express intent, prioritizing cost savings over raw performance or enforcing strict service level objectives (SLOs).[23][26]

Quality of Service (QoS): Specifies workload performance and priority requirements.[23]

- qos.class: latency_critical, best_effort, batch
- slo.latency_p99: <100ms
- priority: high

Efficiency and Cost: Defines scheduler optimization goals.[26]

- objective.cost: minimize (e.g., prefer spot instances).
- objective.power_consumption: <250W (for power-aware scheduling).[26]
- objective.thermal_profile: avoid_hotspots (for thermal-aware scheduling).[28]
- objective.utilization: maximize

Placement Constraints: Enforces explicit rules about where and how tasks can be placed, often for availability or data governance reasons.[23]

- placement.affinity: {colocate_with: service_A}
- placement.anti_affinity: {spread_across: racks}
- placement.constraint: {attribute: rack, op: unique}

Example Policy-Level RDL Specification

```
# Example YAML-like RDL specification for Policy-Level semantics
policy:
  quality_of_service:
    class: latency_critical
    slo:
      latency_p99: "<100ms"
      availability: 99.9
    priority: high
    preemptible: false
  efficiency:
    objective:
      cost: minimize
      power_consumption: "<250W"
      thermal_profile: avoid_hotspots
      utilization: maximize
    budget:
      max_cost_per_hour: "$10"
  placement:
    affinity:
      colocate_with: service_A
    anti_affinity:
      spread_across: racks
    constraint:
      - attribute: rack
        operator: unique
      - attribute: zone
        operator: in
        values: [us-east-1a, us-east-1b]
    preferred_region: us-east
```

3.4 Security-Level Semantics (The "Who and How Safely")

This new dimension addresses critical security and isolation requirements that modern workloads increasingly demand. It acknowledges that sharing detailed hardware information creates potential security implications, and that different workloads have varying trust and isolation needs.

Isolation Requirements: Specifies the degree of physical or logical separation required from other workloads.

- isolation.level: shared, dedicated_cpu, dedicated_host, dedicated_rack
- isolation.tenant_separation: required (indicating workload must not share resources with other tenants)
- isolation.side_channel_protection: required (requesting mitigations for speculative execution attacks)

Compliance and Governance: Defines regulatory or organizational requirements affecting placement.

- compliance.data_residency: {region: eu-west}
- compliance.certification: {pci_dss: required, hipaa: required}
- compliance.data_classification: restricted

Hardware Trust: Specifies requirements for hardware-based security features.

- security.trusted_execution: {type: sgx, attestation: required}
- security.secure_boot: required
- security.hardware_root_of_trust: required

Example Security-Level RDL Specification

```
# Example YAML-like RDL specification for Security-Level semantics
security:
  isolation:
    level: dedicated_host
    tenant_separation: required
    side_channel_protection: required
    noisy_neighbor_mitigation: high
  compliance:
    data_residency:
      region: eu-west
      jurisdiction: EU
    certification:
      - pci_dss: required
      - hipaa: required
    data_classification: restricted
  hardware_trust:
    trusted_execution:
      type: sgx
      attestation: required
    secure_boot: required
    hardware_root_of_trust: required
    confidential_computing: enabled
```

Integrated RDL Example

A complete resource request combining all four dimensions would allow users to express nuanced requirements spanning performance, behavior, policy and security concerns:

```
# Complete YAML-like RDL resource request example
apiVersion: v1
kind: ResourceRequest
metadata:
  name: financial-ml-training-job
  namespace: finance
  labels:
    department: trading
    application: risk-modeling
spec:
  # Hardware-Level Semantics
  hardware:
    compute:
      cpu:
        architecture: x86_64
        isa: [avx2, avx512]
        cores: 64
      accelerator:
        type: nvidia_a100
        count: 8
    memory:
      capacity: 1TB
      bandwidth: ">300GB/s"
      numa:
        policy: strict_local
    storage:
      type: local_nvme
      capacity: 10TB
    interconnect:
      type: nvlink
      topology: fully_connected

  # Workload-Level Semantics
```

```
workload:
  execution_model:
    job:
      type: batch
      structure:
        gang_scheduled: true
        size: 8
    task:
      dependencies: dag
  resource_pressure:
    sensitivity:
      memory_bandwidth: high
      l3_cache: critical
  communication_pattern:
    pattern:
      type: all_reduce
      volume: high
    data:
      locality_preference: strong

# Policy-Level Semantics
policy:
  quality_of_service:
    class: batch
    priority: high
    preemptible: false
  efficiency:
    objective:
      cost: balanced
      power_consumption: "<2000W"
  placement:
    anti_affinity:
      spread_across: racks

# Security-Level Semantics
security:
```

```
isolation:
  level: dedicated_host
  side_channel_protection: required
compliance:
  data_residency:
    region: us-east
  certification:
    - pci_dss: required
  data_classification: confidential
hardware_trust:
  confidential_computing: enabled
```

By structuring resource requests along these four axes, systems capture far more complete pictures of workload needs and contexts.[19] This structured vocabulary provides clear paths away from "cores and gigabytes" ambiguity toward truly intelligent, semantically-aware resource management.[19][27]

mantic Dimension	Category	Example Attribute	Complexity (1-5)	Impact on Performance (1-5)
Hardware-Level	Compute	cpu.isa	3	5
	Memory	numa.policy	4	5
	Storage	disk.type	2	3
	Network	interconnect.topology	4	5
Workload-Level	Execution Model	job.structure	3	4
	Resource Pressure	sensitivity.memory_bandwidth	4	5
	Communication	data.locality_preference	3	4
Policy-Level	QoS	slo.latency_p99	2	3
	Efficiency	objective.power_consumption	3	2
	Placement	placement.anti_affinity	3	4
Security-Level	Isolation	isolation.level	4	3
	Compliance	compliance.data_residency	3	2
	Hardware Trust	security.trusted_execution	5	4

Table 1: Table 1: Multi-Dimensional Resource Semantics Taxonomy

4. A Review of Semantic Scheduling Models

The transition toward richer resource semantics represents not merely a theoretical exercise; it embodies a trend already underway, driven by the practical limitations of the abstract model.[27] Influential schedulers from different domains have, through necessity, developed mechanisms to incorporate deeper semantic understanding into their decision-making processes.[27][28] This section analyzes three major classes of schedulers—HPC, hyperscale, and open-source—along with more recent specialized systems, evaluating them against the taxonomy proposed in Section 3 to understand their strengths and weaknesses.[28]

4.1 The HPC Approach: Explicit Constraints in Slurm

The Slurm Workload Manager stands as the de facto standard scheduler for many of the world's largest supercomputers.[25] The HPC environment features extreme hardware heterogeneity and workloads exquisitely sensitive to performance.[25] Consequently, Slurm was designed from the ground up to handle explicit, fine-grained resource requests.[25]

Semantic Strengths: Slurm's primary strength lies in its comprehensive support for Hardware-Level Semantics.[25] Users are encouraged to provide detailed job specifications, and the scheduler provides powerful primitives to honor them.[25] A user can request nodes with a specific processor architecture (--constraint="cas|mil"), a minimum number of CPUs per node (--mincpus), a specific memory-per-CPU ratio (--mem-per-cpu), or a range of acceptable node counts (--nodes=128-256).[26]

Furthermore, Slurm has native, sophisticated support for topology-aware scheduling.[27] Using the topology/tree plugin, Slurm's basic algorithm identifies the lowest-level switch in the network hierarchy that can satisfy a job's request, thereby minimizing inter-task communication latency by keeping the allocation physically compact.[27] This deep understanding of the network interconnect serves as a first-class feature, reflecting the critical importance of communication performance for large-scale parallel applications.[28]

Semantic Weaknesses: Slurm focuses on a specific execution model: non-preemptive, gang-scheduled batch jobs.[36] Its support for Policy-Level Semantics appears less developed compared to hyperscale systems.[36] While it has a robust priority system that considers factors like job size, age, and fair-share, its default scheduling algorithm fundamentally follows FIFO within priority levels.[36] It lacks the dynamic, SLO-driven preemption mechanisms needed to efficiently co-locate latency-critical services with batch workloads.[36] Its understanding of Workload-Level Semantics largely stays limited to what can be expressed through hardware requirements; it does not, for instance, have a native concept of a workload's sensitivity to memory bandwidth contention.[36]

4.2 The Hyperscale Approach: Declarative SLOs in Borg

Google's Borg functions as a cluster manager designed to operate at immense scale, managing hundreds of thousands of machines running a highly heterogeneous mix of workloads.[37] The primary challenge for Borg involves maximizing the utilization of this massive fleet while simultaneously meeting the stringent SLOs of long-running, latency-sensitive services (e.g., web search, Gmail), which it terms "prod" jobs.[4]

Semantic Strengths: Borg's architecture exemplifies Policy-Level Semantics.[4] Instead of requesting specific hardware, users submit jobs using a declarative specification language that describes the job's requirements and, most importantly, its QoS class and priority.[4] Borg defines non-overlapping priority bands for different types of work (e.g., monitoring, production, batch, best-effort).[4] The scheduler's core mechanism employs fine-grained, priority-based preemption.[4] It aggressively over-commits resources to low-priority "non-prod" jobs, assuming they will not use their full allocation.[4] When a high-priority "prod" job needs resources, Borg immediately preempts (kills) lower-priority tasks to make room.[4] This policy-driven approach allows Borg to achieve exceptionally high cluster utilization.[37] More recent evolutions of Borg include systems like Autopilot, which automatically adjusts a job's resource requests based on its historical usage.[31] This represents a form of learned Workload-Level Semantics, where the system infers a workload's true needs, relieving the user of the burden of precise specification.[31]

Semantic Weaknesses: The user-facing API deliberately abstracts away most Hardware-Level Semantics.[37] While Borg's scheduler maintains internal awareness of machine heterogeneity and makes sophisticated placement decisions, this remains largely hidden from the user.[37] The primary contract between the user and Borg centers on QoS and priority, not on requests for specific hardware features.[37] This design choice prioritizes operational simplicity and fleet-wide optimization over allowing users to micromanage placement.[37][30]

4.3 The Extensible Approach: The Kubernetes Framework

Kubernetes has emerged as the dominant open-source platform for container orchestration.[32] Its design philosophy emphasizes modularity and extensibility, providing a core set of functionalities that can be augmented and adapted to a wide variety of use cases.[32]

Semantic Strengths: The greatest strength of Kubernetes lies in its pluggable scheduling framework.[33] The default scheduler performs a two-phase process of filtering (eliminating nodes that cannot run a pod) and scoring (ranking the feasible nodes).[35] While the default criteria remain basic (e.g., resource requests), the framework was designed for extension.[35] Richer semantics get layered on top of the basic pod specification using a variety of mechanisms.[33]

- nodeAffinity and nodeSelector allow pods to target nodes with specific labels, providing a way to express Hardware-Level Semantics (e.g., disktype: ssd, gpu: true).[34]
- PodAffinity and podAntiAffinity provide control over Policy-Level placement constraints, allowing users to specify that pods should be co-located or spread across topological domains like nodes, racks, or zones.[36]
- Taints and tolerations enable nodes with special hardware (like GPUs) to be reserved for pods that explicitly tolerate the taint, preventing general-purpose workloads from being scheduled on them.[33]

Semantic Weaknesses: As detailed in Section 2.3, the fundamental weakness of Kubernetes stems from its pod-centric model.[6] It lacks a native, first-class concept of a "job," making it inherently ill-suited for managing complex, multi-component AI/ML workloads out of the box.[6] The mechanisms for expressing semantics, while powerful, appear bolted onto the side of the core resource model rather than being integrated into it.[33] This leads to a complex and sometimes unwieldy user experience.[33]

Operational Consequences: This architectural limitation forces organizations to install and manage a complex stack of third-party schedulers and plugins, often with overlapping functionality and inconsistent behaviors. A typical AI/ML platform built on Kubernetes might require multiple custom schedulers such as:

- Volcano or YuniKorn for gang scheduling and fair sharing
- KubeFlow's MPIOperator for distributed training jobs
- GPU-specific plugins like NVIDIA's device plugin
- Custom operators for specialized hardware acceleration
- Additional frameworks for resource quotas and multi-tenancy

This proliferation of schedulers creates significant operational burden:

- Administrators must maintain compatibility between multiple scheduling components during upgrades
- Conflicts between schedulers can lead to unpredictable behavior and deadlocks
- Troubleshooting becomes exponentially more complex as issues may span multiple components
- Users face a steep learning curve, needing to understand the interactions between numerous scheduling abstractions
- Organizations often require specialized platform teams just to manage this complexity, adding substantial operational costs

The lack of native job-level queuing has led to the proliferation of third-party, domain-specific schedulers (e.g., Volcano, YuniKorn, KubeFlow's schedulers) that attempt to retrofit these essential Workload-Level Semantics onto the platform.[6]

4.4 The Specialized Approach: Ant-Man and Hydra for AI/ML

Recognizing the deficiencies of general-purpose schedulers, a new generation of specialized systems has emerged, designed explicitly for AI/ML and large-scale data analytics.[46]

AntMan (Alibaba): AntMan functions as a deep learning infrastructure that co-designs the cluster scheduler with the deep learning framework itself.[46] Its key innovation lies in its deep understanding of Workload-Level Semantics.[46] It recognizes that DL training jobs have fluctuating resource demands and exploits this by dynamically scaling memory and computation allocations.[46] This allows it to co-execute multiple jobs on a shared GPU, using spare resources to improve overall hardware utilization by up to 34-42% without compromising fairness.[46] This represents a tight integration between the scheduler and the application, enabling optimizations impossible with a generic scheduler.[46][38]

Hydra (Microsoft): Hydra serves as a resource manager for data-center scale analytics, designed to handle scheduling decision rates 10-100x higher than general-purpose frameworks.[49] Its primary contribution involves a federated architecture that addresses scalability.[49] A cluster gets divided into loosely coordinating sub-clusters, each with its own local scheduler for task placement.[49] A central Global Policy Generator (GPG) manages Policy-Level Semantics, periodically obtaining an aggregate view of the cluster state and pushing updated scheduling policies to the sub-clusters.[40] This design allows operators to dynamically change the scheduling behavior of a 50,000-node cluster within seconds, providing immense operational agility.[49][40]

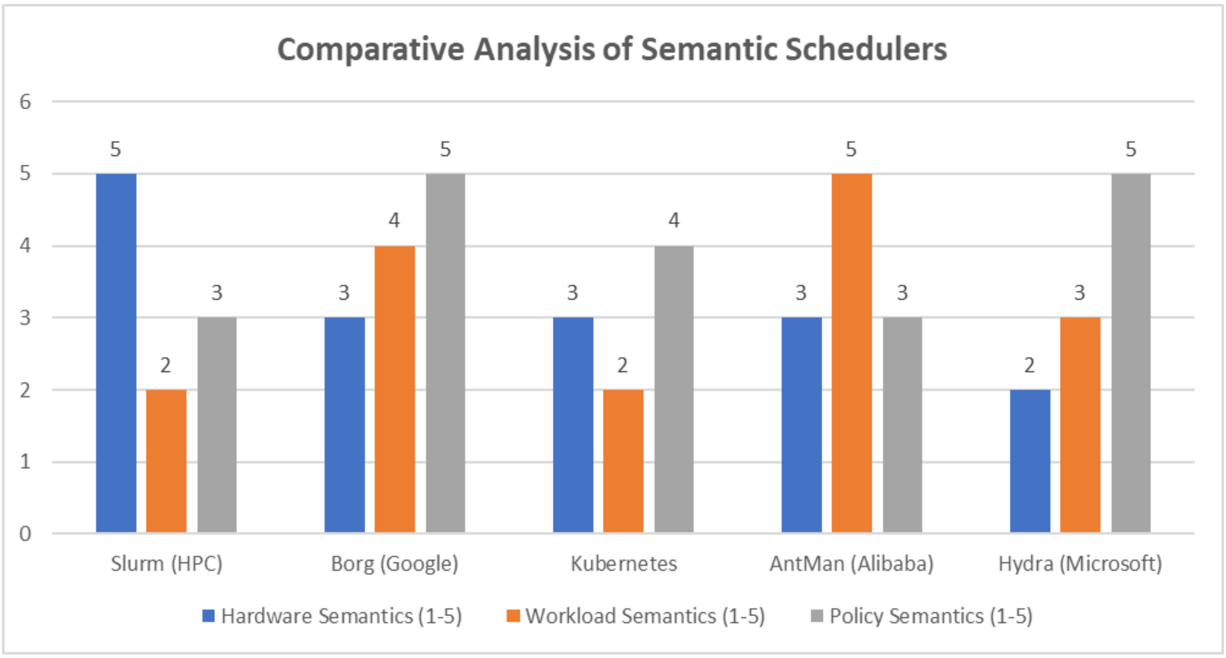

Figure 2: Evolution of Resource Semantics in Modern Schedulers

5. Opportunities and Challenges

The shift towards a richer, semantic model for resource requests opens up significant opportunities for improving the performance, efficiency, and predictability of large-scale clusters.[41] However, this transition comes with substantial technical and conceptual challenges.[41] A fundamental tension emerges between the desire for performance through hardware specialization and the need for portability through abstraction—a tension that has defined the historical divergence between the HPC and cloud computing communities.[41] AI/ML workloads now demand the best of both worlds, forcing a reconciliation of these competing philosophies.[41][42]

5.1 Opportunities

Improved Performance and Predictability: The most direct benefit of semantic scheduling manifests as raw performance.[42] By matching workloads to the micro-architectures best suited for them, schedulers can unlock significant speedups.[42] This includes placing vector-heavy jobs on CPUs with powerful AVX capabilities, ensuring memory-intensive tasks have NUMA-local data access, and scheduling distributed training tasks on nodes with high-bandwidth, low-latency interconnects.[6] This level of matching also reduces performance variability.[42] When a workload consistently gets placed on hardware that meets its semantic needs, its execution time becomes more predictable, which proves critical for meeting SLOs and for accurate capacity planning.[42]

Increased Cluster Utilization: A deeper understanding of workload behavior enables more intelligent co-location, leading to higher cluster utilization.[43] Instead of leaving safety margins to avoid interference, a semantic scheduler can safely pack nodes more densely by pairing complementary workloads.[43] For instance, it could co-locate a memory-bandwidth-intensive job with a compute-intensive, cache-resident job, knowing they will not contend for the same bottleneck resource.[43] This core principle allows systems like Borg to achieve high utilization by mixing latency-sensitive and batch jobs, and serves as the explicit goal of specialized systems like AntMan for GPU sharing.[37][46]

Enhanced Energy Efficiency: By elevating power and thermal properties to first-class semantic concepts, schedulers can directly optimize for energy efficiency.[26] A thermal-aware scheduler can distribute workloads to avoid creating hotspots, thereby reducing the energy consumed by cooling systems, which can represent a substantial fraction of a data center's operational cost.[28] Similarly, a power-aware scheduler can place jobs to stay within a node's or rack's power budget, or consolidate workloads onto fewer machines to allow others to enter low-power states.[26]

5.2 Challenges

API Complexity and Usability

The primary risk of a highly expressive resource description model lies in becoming too complex for the average user.[46] A combinatorial explosion of options can lead to a steep learning curve and a high cognitive burden, potentially resulting in misconfigurations that harm performance rather than help it.[46] The challenge involves designing an API that exposes powerful semantics without sacrificing usability, perhaps through sensible defaults, higher-level abstractions, or automated recommendation systems.[46]

Scheduler Complexity

From the system's perspective, the scheduling problem transforms from a relatively straightforward bin-packing exercise into a complex, multi-objective optimization problem across a high-dimensional feature space.[47] Finding an optimal placement that balances hardware matching, policy enforcement, and QoS requirements proves computationally intensive and often NP-hard.[47] This increased complexity can lead to higher scheduling latency, which may be unacceptable for environments with high task churn.[47] Schedulers will need to rely on sophisticated heuristics, approximation algorithms, or machine learning models to make good decisions promptly.[47]

Developer Experience and Workflow Integration

While the paper has primarily focused on the scheduler and operator perspectives, the shift to semantic resource requests creates significant implications for application developers, a key stakeholder group often overlooked in discussions of resource management. Compared to the simplicity of requesting "2 CPUs and 8GB RAM," specifying detailed semantic requirements demands a much deeper understanding of application behavior and hardware interactions.

This semantic shift impacts the developer workflow in several critical ways:

- Workload Characterization Burden: Developers must now profile and characterize their applications across multiple dimensions—memory access patterns, cache sensitivity, communication topology, and more. This requires specialized expertise many developers lack and adds significant overhead to the development process.
- CI/CD Pipeline Complexity: Modern application delivery relies on automated CI/CD pipelines. Semantic resource requests complicate these pipelines by introducing environment-specific dependencies that are difficult to abstract. Testing becomes more complex when applications require specific hardware features, potentially necessitating multiple test environments to ensure compatibility across deployment targets.
- Multi-environment Deployment Challenges: Organizations often deploy applications across development, testing, staging, and production environments, as well as across multiple cloud providers. Semantic resource requests that work optimally in one environment may not translate directly to another, creating a matrix of configurations that developers must maintain.

- Documentation and Knowledge Transfer: Teams must document not just what their application does, but how it interacts with underlying hardware. This represents a significant shift in the traditional separation of concerns between application developers and infrastructure teams.

Addressing these challenges requires new developer-focused tools and methodologies:

- Automated Profiling Tools: Integrated development environments need built-in capabilities to analyze applications and suggest optimal semantic resource specifications.
- Semantic Request Generators: ML-based systems that can observe application behavior during testing and automatically generate appropriate semantic requests.
- Abstraction Libraries: Frameworks that allow developers to express intent (e.g., "this component is memory-intensive") rather than specific hardware requirements.
- Verification and Simulation Tools: Systems that can validate semantic requests against available infrastructure before deployment and predict performance implications.

Without addressing the developer experience, even the most sophisticated semantic scheduling system risks low adoption rates or incorrect usage. The success of this paradigm shift depends not just on scheduler capabilities, but on making semantic resource specification accessible and intuitive for the developers who must ultimately integrate it into their applications and workflows.

Portability and Standardization

The "cores and gigabytes" model, for all its flaws, offers one crucial benefit: portability.[48] A container image defined with abstract resource requests can, in theory, run on any cloud provider.[48] A semantic request, however, that specifies a particular hardware feature (e.g., accelerator.type: nvidia_a100) remains inherently non-portable.[48] This creates a significant challenge for multi-cloud and hybrid-cloud strategies, as it risks tying workloads to a specific vendor's hardware.[48] A successful semantic framework must resolve this tension.[48] It cannot be purely physical, as this would sacrifice portability, nor can it be purely abstract, as this would sacrifice performance.[48] The solution likely lies in allowing users to specify intent (e.g., high_interconnect_bandwidth: true), which each platform's scheduler can then map to the best available local hardware (e.g., NVLink on one platform, Infinity Fabric on another).[48][49] This approach preserves the semantic goal while allowing for platform-specific implementation.[49]

Multi-Cloud Resource Federation

Extending semantic scheduling across multiple cloud providers introduces another layer of complexity.[50] True resource federation gets hampered by a lack of visibility into other providers' infrastructure, inconsistent security and compliance models, and, most importantly, incompatible APIs and data formats.[50] Without a shared, standardized semantic understanding of resources, a meta-scheduler cannot make intelligent, performance-aware placement decisions across cloud boundaries.[50] It cannot compare an AWS instance with a specific networking capability to a GCP instance with a different but functionally similar one, making true optimization across federated resources an unsolved problem.[50][51]

Category	Factor	Benefit/Challenge Level (1-5)	Implementation Complexity (1-5)	Business Impact (1-5)
Opportunity	Performance Predictability	5	3	5
	Cluster Utilization	4	4	5
	Energy Efficiency	3	3	4
Challenge	API Complexity	4	5	3

	Scheduler Complexity	5	5	4
	Portability Issues	4	4	5
	Multi-Cloud Federation	5	5	4

Table 2: Opportunities and Challenges in Semantic Resource Scheduling

6. Future Directions

Addressing the challenges outlined above requires a concerted effort from the systems research community.[52] The path forward involves not only building more intelligent schedulers but also establishing the common languages and frameworks necessary for a portable, interoperable, and semantically rich ecosystem.[52] Two key areas of research promise particular fruitfulness: the standardization of a resource description language and the application of Large Language Models to the scheduling problem itself.[52][53]

6.1 Towards a Standard Resource Description Language (RDL)

The current landscape of resource specification remains fragmented.[49] Each scheduler and cloud provider uses its proprietary mechanisms, creating a "Tower of Babel" that impedes portability and interoperability.[49] The development of a standard Resource Description Language (RDL) represents a critical step toward resolving this.[49] Such a standard would provide a common, unambiguous way for users to express the rich requirements of their workloads.[49]

This problem is not new.[49] The industry has seen similar standardization efforts with languages for describing web services, such as WSDL (Web Services Description Language), and for defining cloud service topologies, like TOSCA (Topology and Orchestration Specification for Cloud Applications).[49] Lessons can also be drawn from the semantic web community's work on structured data representation, such as RDF (Resource Description Framework), which uses triples to create machine-readable statements about resources.[55]

A future RDL should be built upon a formal, layered taxonomy, such as the one proposed in Section 3.[19][49] It should allow users to specify requirements at different levels of abstraction, from high-level policy intent (e.g., objective: minimize_cost) down to specific hardware attributes (e.g., cpu.isa: avx512).[49] This layered approach would help manage complexity and resolve the tension between performance and portability.[49] A user could specify a portable, intent-based request that any compliant scheduler could satisfy, or provide a more detailed, hardware-specific profile to unlock maximum performance on a particular platform.[49] A standard RDL would serve as the foundational technology enabling true multi-cloud resource federation and a competitive, open ecosystem of semantically-aware schedulers.[49][50]

A Concrete Path to RDL Standardization

While the need for standardization is clear, the path to achieving it requires specific, actionable steps:

Establish a Neutral Working Group: A viable path to standardization would begin with establishing a dedicated working group within a neutral industry foundation like the Cloud Native Computing Foundation (CNCF). This group should bring together key stakeholders including:

- Cloud service providers (AWS, Google Cloud, Microsoft Azure)
- Hardware manufacturers (Intel, AMD, NVIDIA, ARM)
- Enterprise end-users with significant ML/AI investments
- Open-source project maintainers from Kubernetes, Slurm, and specialized ML schedulers
- Academic researchers specializing in resource management

Develop a Phased Approach: The standardization effort should follow a phased approach:

- Phase 1 (6-12 months): Develop a core specification focusing on the most critical semantic dimensions and attributes with proven impact

- Phase 2 (12-18 months): Create reference implementations for major platforms (Kubernetes, cloud providers)
- Phase 3 (18-24 months): Expand the specification to include advanced semantics like security isolation and energy efficiency
- Phase 4 (24+ months): Establish compliance testing and certification processes

Leverage Existing Standards: The RDL should build upon and integrate with existing standards rather than competing with them:

- Incorporate relevant portions of the Kubernetes resource model
- Align with the Open Compute Project's hardware specifications
- Ensure compatibility with existing infrastructure-as-code tools like Terraform and Pulumi
-

Ensure Backward Compatibility: A critical success factor will be allowing incremental adoption without forcing organizations to completely rewrite their existing resource specifications. The standard should provide clear migration paths and translation mechanisms between legacy resource models and the new RDL.

This concrete standardization path recognizes that success requires both technical excellence and industry adoption, necessitating a collaborative, phased approach with clear governance and backward compatibility guarantees.

6.2 LLM-driven Scheduling

The complexity of both the scheduling problem and the user-facing API presents an opportunity for a new class of AI-driven solutions.[57] Large Language Models (LLMs) are emerging as a promising technology for tackling complex reasoning and optimization tasks, and recent research has begun to explore their application to cluster scheduling.[57]

Intent Translation: LLMs could serve as a natural language interface to a complex semantic scheduling API, addressing the usability challenge.[57] A user could state their goal in plain English, such as, "Run this distributed training job for my new vision model as fast as possible, but prioritize using cheaper spot instances if the completion time won't increase by more than 10%."[57] An LLM-based agent could then translate this high-level intent into a detailed, formal RDL specification, selecting the appropriate hardware, workload, and policy semantics without requiring the user to master the complex API.[57]

Multi-Objective Optimization: The scheduling decision itself represents a multi-objective optimization problem that often proves too complex for traditional solvers to handle in real-time.[70] LLMs, particularly when coupled with reasoning frameworks like ReAct (Reason + Act), can be prompted to reason about the trade-offs between competing objectives—such as performance, cost, fairness, and energy efficiency—in a more flexible and human-like way than hard-coded heuristics.[70] The LLM could receive the current cluster state, the incoming job's semantic request, and a set of policy goals, and then generate a reasoned placement decision along with a natural language explanation for its choice.[57][70]

Dynamic Adaptation and Learning: A key challenge in large-scale systems stems from constantly changing workload patterns and cluster conditions.[57] An LLM-based scheduler could be designed to learn from historical telemetry data.[57] By analyzing past scheduling decisions and their resulting performance outcomes, the model could learn complex correlations—for example, which types of workloads interfere with each other on specific hardware platforms—and dynamically adapt its scheduling policies over time.[57] This would enable the scheduler to continuously improve its performance in a way that would be infeasible to achieve with manually tuned, static algorithms.[57][69]

Risks and Challenges of LLM-driven Scheduling

While LLM-driven scheduling offers exciting possibilities, several critical risks and challenges must be addressed before these systems can be deployed in production environments:

Hallucination and Non-determinism: LLMs are known to occasionally generate plausible-sounding but incorrect outputs, a phenomenon known as "hallucination." In a scheduling context, this could lead to:

- Generating invalid resource specifications that appear correct

- Recommending incompatible hardware configurations
- Creating impossible placement decisions based on misunderstood constraints
- Producing inconsistent decisions for identical inputs over time

Explainability and Auditability: Production schedulers require transparent decision-making that can be audited and verified, especially in regulated industries. LLM-based systems must provide:

- Clear explanations for why specific placement decisions were made
- Audit trails showing the reasoning process
- Mechanisms to trace how specific inputs influenced the final decision
- Compliance with governance requirements for automated decision-making

Safety Guardrails: LLM-driven schedulers require robust safety mechanisms to prevent harmful decisions:

- Validation layers that verify all LLM-generated specifications against known constraints
- Circuit breakers that revert to traditional scheduling algorithms if abnormal patterns are detected
- Simulation environments to test decisions before implementation
- Human-in-the-loop approval flows for high-risk or high-value workloads

Performance and Latency Concerns: Current LLMs have significant inference latency, potentially introducing unacceptable delays in scheduling decisions:

- Scheduling latency is a critical metric in high-throughput systems
- Adding hundreds of milliseconds per decision could severely impact cluster performance
- Optimization techniques like distillation, caching, and specialized hardware are needed

Training Data Biases: LLMs trained on historical scheduling decisions may perpetuate and amplify existing inefficiencies:

- Models might learn to reproduce human biases and suboptimal patterns
- Continuous evaluation against objective metrics is essential
- Synthetic data generation may be needed to cover rare edge cases

A viable approach would be to implement LLM-driven scheduling in phases, starting with non-critical advisory roles (suggesting optimizations to human operators) before gradually increasing autonomy as confidence in the system grows.[] Hybrid architectures that combine traditional algorithmic approaches with LLM components may offer the best balance of innovation and reliability in the near term.[]

By acknowledging and addressing these challenges directly, the research community can develop LLM-driven scheduling systems that deliver on their transformative potential while maintaining the reliability and trustworthiness that production environments require.

Conclusion

The paradigm of requesting computational resources by simply counting CPU cores and measuring memory in gigabytes has been remarkably successful, underpinning the growth of cloud computing for over a decade. However, its utility has reached its limit. For the modern, performance-critical workloads that now drive innovation in fields like AI and data science, this coarse-grained abstraction is no longer sufficient. It obscures vital information about the underlying hardware and application behavior, leading to systems that are inefficient, unpredictable, and difficult to optimize. The paper argues for a fundamental shift in resource management approaches—moving away from simple counting toward a rich, semantic understanding of both resources and workloads. A systematic deconstruction of the current model's failure modes reveals issues from CPU core non-fungibility to hidden contention on shared resources like memory bandwidth and cache. Addressing these shortcomings requires a formal taxonomy of resource semantics, providing a structured vocabulary to describe the "what," "how," and "why" of resource requests. Reviews of influential schedulers like Slurm, Borg, and Kubernetes reveal that this semantic shift is already happening in practice, albeit in a fragmented and system-specific manner. The path forward requires embracing complexity, with immense opportunities for more predictable performance, higher cluster utilization, and greater energy efficiency. Yet significant challenges exist across API design, scheduler complexity, and the tension between portability and performance. Realizing this future demands investment in next-generation semantically-aware schedulers and collaboration on standardization efforts, such as developing a common Resource Description Language, ensuring large-scale computing becomes not only high-performance but also open, portable, and interoperable. The era of simply counting cores and gigabytes has ended; the era of semantic resource management has begun.

References

[1] Cody Slingerland, "The Simple Guide To The History Of The Cloud," CloudZero, 2023. [Online]. Available: https://www.cloudzero.com/blog/history-of-the-cloud/

[2] GeeksforGeeks, "Evolution of Cloud Computing," 2025. [Online]. Available: https://www.geeksforgeeks.org/cloud-computing/evolution-of-cloud-computing/

[3] Abhishek Verma et al., "Large-scale cluster management at Google with Borg," ACM, 2015. [Online]. Available: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43438.pdf

[4] Cloud Native Computing Foundation, "Why the Kubernetes Scheduler Is Not Enough for Your AI Workloads," 2020. [Online]. Available: https://www.cncf.io/blog/2020/08/10/why-the-kubernetes-scheduler-is-not-enough-for-your-ai-workloads/

[5] Nimbus, "Struggling with AI/ML on Kubernetes? Why Specialized Schedulers Are Key to Efficiency," Medium, 2025. [Online]. Available: https://medium.com/@nimbus-nimo/struggling-with-ai-ml-on-kubernetes-why-specialized-schedulers-are-key-to-efficiency-e626271b30d2

[6] Chaim Rand, "The Crucial Role of NUMA Awareness in High-Performance Deep Learning," Medium, 2025. [Online]. Available: https://chaimrand.medium.com/the-crucial-role-of-numa-awareness-in-high-performance-deep-learning-99ae3e8eb49a

[7] Di Xu, Chenggang Wu, and Pen Chung Yew, "On mitigating memory bandwidth contention through bandwidth-aware scheduling," University of Minnesota. [Online]. Available: https://experts.umn.edu/en/publications/on-mitigating-memory-bandwidth-contention-through-bandwidth-aware

[8] Broadcom, "Impact of virtual machine memory and CPU resource limits," 2025. [Online]. Available: https://knowledge.broadcom.com/external/article/326270/impact-of-virtual-machine-memory-and-cpu.html

[9] James R Reinders, "Intel® AVX-512 Instructions," Intel, 2017. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html

[10] Mathias Gottschlag, Philipp Machauer, Yussuf Khalil, and Frank Bellosa, "Fair Scheduling for AVX2 and AVX-512 Workloads," in the Proceedings of the 2021 USENIX Annual Technical Conference, 2021. [Online]. Available: https://www.usenix.org/system/files/atc21-gottschlag.pdf

[11] Sean Broestl, "Minimizing NUMA Effects on Machine Learning Workloads in Virtualized Environments," Harvard Library, 2021. [Online]. Available: https://dash.harvard.edu/server/api/core/bitstreams/6647d197-5a8c-4331-9542-7f752ff3d70d/content

[12] Ming Liu and Tao Li, "Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads," ResearchGate, 2014. [Online]. Available: https://www.researchgate.net/publication/271472618_Optimizing_virtual_machine_consolidation_performance_on_NUMA_server_architecture_for_cloud_workloads

[13] Probir Roy et al., "NUMA-Caffe: NUMA-Aware Deep Learning Neural Networks," ACM, 2018. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/ai/documents/NUMA-Caffe.pdf

[14] Scott Rixner et al., "Memory Access Scheduling," Stanford University. [Online]. Available: http://cva.stanford.edu/publications/2000/mas.pdf

[15] Evangelos Koukis and Nectarios Koziris, "Memory and network bandwidth aware scheduling of multiprogrammed workloads on clusters of SMPs," SciSpace, 2006. [Online]. Available: https://scispace.com/pdf/memory-and-network-bandwidth-aware-scheduling-of-2w8nhqd7hj.pdf

[16] Christina Delimitrou and Christos Kozyrakis, "IBench: Quantifying interference for datacenter applications," ResearchGate, 2013. [Online]. Available: https://www.researchgate.net/publication/261486342_IBench_Quantifying_interference_for_datacenter_applications

[17] Nan Guan et al., "Cache-Aware Scheduling and Analysis for Multicores," 2009. [Online]. Available: https://embedded.cs.uni-saarland.de/literature/CacheAwareSchedulingAndAnalysisForMulticores.pdf

[18] Guillaume Aupy et al., "Co-scheduling algorithms for cache-partitioned systems," Inria. [Online]. Available: https://people.bordeaux.inria.fr/gaupy/ressources/pub/confs/apdcm2017_cache.pdf

[19] Stack Overflow, "A simple example of a cache-aware algorithm?". [Online]. Available: https://stackoverflow.com/questions/473137/a-simple-example-of-a-cache-aware-algorithm

[20] Marcelo Amaral et al., "Topology-Aware GPU Scheduling for Learning Workloads in Cloud Environments," 2017. [Online]. Available: https://core.ac.uk/download/pdf/157810919.pdf

[21] Ching-Hsiang Chu et al., "NV-Group: Link-Efficient Reduction for Distributed Deep Learning on Modern Dense GPU Systems," ACM, 2020. [Online]. Available: https://par.nsf.gov/servlets/purl/10198156

[22] Reddit, "Why the Default Kubernetes Scheduler Struggles with AI/ML Workloads (and an Intro to Specialized Solutions),". [Online]. Available: https://www.reddit.com/r/kubernetes/comments/1jswmvy/why_the_default_kubernetes_scheduler_struggles/

[23] Abdolkhalegh Baluch et al., "Thermal and Power-Aware VM Scheduling on Cloud Computing in Data Center," ResearchGate, 2019. [Online]. Available: https://www.researchgate.net/publication/342117805_Thermal_and_Power-Aware_VM_Scheduling_on_Cloud_Computing_in_Data_Center

[24] HPC Core Facility, "What is Slurm, and how do I write and submit a Slurm job?". [Online]. Available: https://hpc.ucdavis.edu/faq/slurm

[25] NASA Center for Climate, "Slurm Best Practices on Discover,". [Online]. Available: https://www.nccs.nasa.gov/nccs-users/instructional/using-slurm/best-practices

[26] Slurm Workload Manager, "Topology Guide." [Online]. Available: https://slurm.schedmd.com/topology.html

[27] Yiannis Georgiou, "Slurm Advanced Scheduling," Ryax. [Online]. Available: https://ust4hpc.sciencesconf.org/data/pages/Slurm_UST4HPC_final.pdf

[28] Yiannis Georgiou et al., "Topology-aware resource management for HPC applications," Teratec. [Online]. Available: https://teratec.eu/COLOC/library/Topology-aware-resource-mgt_RR-8859_.pdf

[29] Morries Jette and Mark Grondona, "SLURM: Simple Linux Utility for Resource Management," Slurm Workload Manager, 2003. [Online]. Available: https://slurm.schedmd.com/slurm_design.pdf

[30] Muhammad Tirmazi et al., "Borg: The Next Generation," 2020. [Online]. Available: https://www.cs.cmu.edu/~harchol/Papers/EuroSys20.pdf

[31] Ravi Gadde, "Architecture," GitHub. [Online]. Available: https://github.com/ravigadde/kube-scheduler/blob/master/docs/design/architecture.md

[32] Kubernetes, "Scheduling Framework," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

[33] Ron Sobol, "A Deep Dive into Kubernetes Scheduling," The New Stack, Nov 30th, 2020. [Online]. Available: https://thenewstack.io/a-deep-dive-into-kubernetes-scheduling/

[34] CloudBolt, "Kubernetes Pod Scheduling: Tutorial and Best Practices," 2025. [Online]. Available: https://www.cloudbolt.io/kubernetes-pod-scheduling/

[35] Kunal Verma, "Kubernetes Scheduling - The Complete Guide," Kubesimplify, 2024. [Online]. Available: https://blog.kubesimplify.com/kubernetes-scheduling-the-complete-guide

[36] Roman Glushach, "Kubernetes Scheduling: Understanding the Math Behind the Magic," Medium, 2023. [Online]. Available: https://romanglushach.medium.com/kubernetes-scheduling-understanding-the-math-behind-the-magic-2305b57d45b1

[37] Wencong Xiao et al., "AntMan: Dynamic Scaling on GPU Clusters for Deep Learning," USENIX. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/xiao

[38] Carlo Curino et al., "Hydra: A federated resource manager for data-center scale analytics," USENIX. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/curino

[39] Rutwik Jain et al., "PAL: A Variability-Aware Policy for Scheduling ML Workloads in GPU Clusters," arXiv:2408.11919v1, 2024. [Online]. Available: https://arxiv.org/html/2408.11919v1

[40] Dingyu Yang et al., "Mitigating Interference of Microservices with a Scoring Mechanism in Large-scale Clusters," arXiv:2407.12248v1, 2024. [Online]. Available: https://arxiv.org/html/2407.12248v1

[41] Xiaolong Zhang et al., "Zeus: Improving Resource Efficiency via Workload Colocation for Massive Kubernetes Clusters," SciSpace, 2017. [Online]. Available: https://scispace.com/pdf/zeus-improving-resource-efficiency-via-workload-colocation-211fimqw0a.pdf

[42] Van Damme and Tobias, "Thermal-aware job scheduling in data centers [0.2em] An optimization approach," University of Groningen, 2019. [Online]. Available: https://pure.rug.nl/ws/portalfiles/portal/86549605/Complete_thesis.pdf

[43] Romil Bhardwaj et al., "ESCHER – Expressive Scheduling with Ephemeral Resources," University of California. [Online]. Available: https://romilb.com/files/ESCHER_SOCC_Poster.pdf

[44] Saumya Mathkar et al., "Scheduling Big Machine Learning Tasks on Clusters of Heterogeneous Edge Devices," ResearchGate, 2025. [Online]. Available: https://www.researchgate.net/publication/389185352_Scheduling_Big_Machine_Learning_Tasks_on_Clusters_of_Heterogeneous_Edge_Devices

[45] Luo Shengmei, "Cloud Computing Standardization and Research," ZTE, 2011. [Online]. Available: https://www.zte.com.cn/global/about/magazine/zte-technologies/2011/3/en_539/235160.html

[46] Falak Nawaz et al., "Service description languages in cloud computing: State-of-the-art and research issues," ResearchGate, 2019. [Online]. Available: https://www.researchgate.net/publication/333530448_Service_description_languages_in_cloud_computing_State-_of-the-art_and_research_issues

[47] Storware, "Top 5 Challenges of Protecting Multi-Cloud Environment,". [Online]. Available: https://storware.eu/blog/top-5-challenges-of-protecting-multi-cloud-environment/

[48] Tenable, "Multi-cloud and hybrid cloud security challenges," 2025. [Online]. Available: https://www.tenable.com/cybersecurity-guide/learn/multi-cloud-and-hybrid-cloud-security

[49] Taylor Karl, "Multi-Cloud Challenges: Best Practices and Strategies," New Horizons, 2024. [Online]. Available: https://www.newhorizons.com/resources/blog/multi-cloud-challenges

[50] Dhruv Seth et al., "Navigating the Multi-Cloud Maze: Benefits, Challenges, and Future Trends," IJGIS, 2024. [Online]. Available: https://ijgis.pubpub.org/pub/plmsrs5y/release/1

[51] Cloud Defense, "What is Web Services Description Language in AWS? Detailed Explanation,". [Online]. Available: https://www.clouddefense.ai/glossary/aws/web-services-description-language

[52] W3C, "RDF 1.2 Concepts and Abstract Syntax," 2025. [Online]. Available: https://www.w3.org/TR/rdf12-concepts/

[53] Enterprise Knowledge, "The Resource Description Framework (RDF)," 2025. [Online]. Available: https://enterprise-knowledge.com/the-resource-description-framework-rdf/

[54] Cesar Miguelañez, "How Task Scheduling Optimizes LLM Workflows," 2025. [Online]. Available: https://latitude-blog.ghost.io/blog/how-task-scheduling-optimizes-llm-workflows/

[55] Zeyu Zhang and Haiying Shen, "PecSched: Preemptive and Efficient Cluster Scheduling for LLM Inference," arXiv:2409.15104v2. [Online]. Available: https://arxiv.org/html/2409.15104v2

[56] Prachi Jadhav, Hongwei Jin, Ewa Deelman, and Prasanna Balaprakash, "Evaluating the Efficacy of LLM-Based Reasoning for Multiobjective HPC Job Scheduling," arXiv:2506.02025, 2025. [Online]. Available: https://arxiv.org/abs/2506.02025

[57] Eran Malach et al., "Don't Stop Me Now: Embedding Based Scheduling for LLMs," ResearchGate, 2024. [Online]. Available: https://www.researchgate.net/publication/384599133_Don't_Stop_Me_Now_Embedding_Based_Scheduling_for_LLMs