Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/

Research Article

Unified Workload Management Through Kubernetes-Native APIs

Shruthi Karpur

Broadcom Inc., USA

ARTICLE INFO

Received: 13 Jan 2026

Revised: 16 Jan 2026

ABSTRACT

Enterprise IT organizations must bridge the architectural gaps between their legacy virtual machine workloads and their cloud-native container-based applications. Virtual machine workloads depend on isolation for reasons such as operating system dependencies, regulatory compliance, or other characteristics such as performance, and microservice architectures are typically deployed on a container orchestration layer for agility and scalability. This article presents unified workload management patterns for declarative, Kubernetes-style deployment and lifecycle management of virtual machines (VMs) along with container workloads on a common control plane. It also presents an architecture that can support key enterprise workload patterns such as workload portability, multi-tenancy governance, developer self-service, and operating model portability across heterogeneous VM and container compute infrastructures. Drawing on experience of running production workloads on both VMs and containers on the same infrastructure, this article shows how Kubernetes native APIs can be extended to provision, configure, and manage VMs in the same declarative way as containers, breaking down infrastructure silos, preserving enterprise security and compliance, and allowing organizations to run faster without sacrificing operational stability. This helps enterprises achieve their digital transformation goals while also protecting existing virtualization investments.

Keywords: Unified Workload Management, Kubernetes-Native APIs, Virtual Machine Orchestration, Container Integration, Enterprise Infrastructure Modernization

1. Introduction and Infrastructure Convergence Challenges

Enterprise computing environments today are based on two decades of parallel evolution, with virtualization starting in the early 2000s, and containerization with the general release of the Docker project in 2013 [1]. This has led to operational complexity in enterprises that run both virtual machine environments and orchestration systems to host both legacy applications and cloud-native applications. The intersection of these models is a watershed moment for enterprises, exposing the impact of fragmented infrastructure and accrued technical debt on their agility and capacity for innovation.

Typically, an enterprise data center will have multiple management platforms, hypervisor-based for VMs and based on Kubernetes for containers. This results in the duplication of operational processes, monitoring frameworks, and operational knowledge in the operations team for both environments. Infrastructure silos are not just technical artifacts. They are organizational barriers that obstruct the sharing of resources, complicate capacity planning problems, and lead to different deployment experiences for application teams. In traditional environments, a developer requesting a VM must often submit a ticket to a central infrastructure team, leading to "wait states" that can last days or weeks. This manual bottleneck is incompatible with the high-velocity requirements of modern CI/CD pipelines. Infrastructure silo operational costs include not only the cost to maintain infrastructure, but the training cost, the proliferation of tooling, as well as delays incurred by context-switching.

Modern data centers require unified control planes for workload deployment, lifecycle management, and policy management, for both economic and technical reasons, across heterogeneous pools of

computing resources. This allows for optimal resource utilization and operational efficiency, as well as simplified management of workloads distributed across multiple computing resources. Unified management approaches seek to achieve the elimination of redundant tooling, the streamlining of organizational processes, and allow cross-functional teams to have an identical workflow across substrates.

Legacy applications may need VM isolation where container environments do not support the legacy application's operating system version, kernel, or required compliance certification. Legacy applications usually cannot be easily containerized without heavy refactoring. This is another reason VM infrastructure continues to be popular. On the other hand, container orchestration provides several benefits for cloud-native applications, such as fast scaling, resource-efficient workloads, and declarative configuration management as in modern DevOps workflows [2]. The paper also compares some architectural design patterns and implementations of unified workload management systems that support both deployment models using Kubernetes-native APIs and bring similar semantics to all compute resources of a cloud provider.

2. Architectural Foundations of Unified Workload Management

Kubernetes defaults to a single workload management framework due to experience with cluster management frameworks over decades, and principles derived from Google's massive internal infrastructure system. These principles include declarative configuration management, reconciliationbased updates and control loops, and the philosophy of managing all resources from an entirely APIoriented perspective. These foundational principles allow Kubernetes to be extended beyond the container-based infrastructure for which it was designed to support VM-based workloads without introducing new architectural concepts. This is accomplished by extending the Kubernetes control plane via an API server pattern that allows registering, validating, and persisting extensible resources (custom resources) in its distributed key/value store, and thereby providing custom domain abstractions[3].

A critical architectural advantage here is the Separation of Concerns. The architecture facilitates a distinct boundary between the Infrastructure Provider (responsible for the physical fabric and hypervisor) and the Tenant Administrator (responsible for the logical organization and users).

Custom Resource Definitions allow declaring the specification of a virtual machine as a first-class object in Kubernetes, so that virtual machines can be managed declaratively like any other Kubernetes containerized workload. The VM specification schema, as part of the CRDs, is expressed declaratively inside the Kubernetes manifest specification that includes compute resources, storage attachments, virtual network interfaces, and boot specifications. Furthermore, virtual machine provisioning via APIs and calls also supports the same request-response cycle and authentication and authorization policies as container provisioning, thereby creating symmetry of operations. Management of the virtual machine lifecycle is done by using Kubernetes operator patterns with desired state specifications defined by the user as a constantly monitored loop. These hypervisor operators communicate with their hypervisor infrastructures via abstraction layers that translate Kubernetes API requests into hypervisor operations.

Hypervisor and container orchestration layers provide abstractions that provide a uniform view and API over the virtualization layer. The performance of resource management under Kubernetes has been studied, highlighting the importance of scheduling algorithms and resource allocation for heterogeneous workloads [4]. Storage abstractions are represented by Container Storage Interface implementations that provide equivalent persistent volumes for VMs as for containerized stateful applications. VM networking is provided likewise by extending CNI plugins to provide an interface for attaching a virtual machine network, enabling the same network policy to be applied across heterogeneous workloads. Multi-workload control plane architecture with specialized admission controllers that deploy per workload type to normalize resource requests, enforce quota policies, inject

security profiles across workloads, and achieve operational consistency while ensuring isolation across the entire application portfolio.

Component	Function	Implementation Mechanism	Workload Support
Custom Resource Definitions	VM specification as first-class objects	Declarative manifest schemas	Virtual machines and containers
Kubernetes Operators	Lifecycle management through reconciliation	Desired state monitoring loops	Heterogeneous workloads
API Server Framework	Resource registration and validation	Distributed key/value store	Custom domain abstractions
Admission Controllers	Resource validation and policy injection	Per-workload type deployment	Multi-workload environments
Storage Abstraction (CSI)	Persistent volume provisioning	Container Storage Interface	VMs and stateful applications
Network Abstraction (CNI)	Network attachment and policy	Container Network Interface plugins	Cross-workload connectivity

Table 1: Kubernetes Control Plane Extension Components for Unified Workload Management [3, 4]

3. Virtual Machine Provisioning Through Declarative APIs

Instead of imperative infrastructure management, API-based virtual machine provisioning creates resources declaratively. VM class definitions can specify compute, memory, and device resources, organized into resource profiles that standardize resource allocation. Classes define templates for VM configuration and admin specific constraints to the VM being deployed, allowing a consistent approach to distributing capacity across organizational boundaries. Classes abstract hardware differences as resource requirements for the application and relate hardware requirements to available infrastructure resources, allowing application teams to request compute resources without needing to understand data center topology. Each virtual machine has a resource allocation policy, which defines quality of service rules of guaranteed, burstable, and best effort based on resource limit specifications and organizational requirements.

Registry patterns are a component of template management systems that deploy golden images containing VM (virtual machine) disk images to heterogeneous locations. VM workload is declaratively specified using manifests that describe operating system template images, resource allocations, network provisioning, and persistent storage requests in a common schema-based format. Microservices architectures have favored declarative infrastructure patterns where individual components are specified in code and version-controlled through specifications rather than handwritten configuration files [5]. API-driven provisioning processes eliminate the manual steps of provisioning infrastructure. Reconciliation controllers also regulate and transition virtual machines through the various stages of the infrastructure lifecycle, making sure that the infrastructure state matches the intended configuration. This enables organizations to replicate your deployment scenarios in development, staging, and production environments, while offering audit trails and rollback capability.

Container Network Interface plugins extend the Kubernetes networking model to virtual machines, allowing virtual machines to connect directly to the container network. Virtual interfaces for virtual

machines can be configured using network attachment definitions, a custom resource type that allows for interface specifications and addressing/policy configuration using Kubernetes-native constructs. An analysis of container orchestration platforms demonstrated that resource allocation and scheduling are important for running heterogeneous workloads on a distributed cluster infrastructure [6]. Persistent data volumes are made available through implementations of the Container Storage Interface that create block storage volumes for VMs. Additional features include dynamic volume provisioning, snapshots, and storage classes. Cloud-init, lunixPrep and sysprep or a similar technology is used to provide SSH keys, user accounts, network settings, and other customization properties to the VM.

Provisioning Element	Purpose	API Pattern	Configuration Method
VM Class Definitions	Standardized resource profiles	Quality-of-service specifications	Compute, memory, device templates
Image Management	Golden image distribution	Registry-based deployment	OS templates and disk images
Declarative Manifests	Infrastructure-as-code specifications	YAML/JSON schemas	Resource and network configurations
Reconciliation Controllers	Automated lifecycle transitions	Continuous state alignment	Development to production pipelines
Network Attachment Definitions	VM network configuration	CNI plugin extensions	Interface and addressing specifications
Persistent Storage Provisioning	Dynamic volume allocation	CSI implementations	Block and file storage volumes
Cloud-init Integration	Runtime configuration injection	Immutable infrastructure patterns	SSH keys, accounts, and application settings

Table 2: Declarative Virtual Machine Provisioning Workflow Elements [5, 6]

4. Multi-Tenancy, Governance, and Security Considerations

Multi-tenancy in unified workload management systems requires proper mechanisms of isolation. These mechanisms ensure unauthorized access and interference between tenant workloads does not cross organization boundaries. Namespace-based isolation for virtual machine workloads utilizes Kubernetes logical partitioning to carve out separate administrative domains where team members operate independently and within their resource boundaries. The principles of isolation outlined above apply to network, storage, and compute resources in addition to naming, so that virtual machines owned by one tenant cannot see or affect another. Role-based access control concepts are applied to the whole workload, with fine-grained permission models controlling what entities or users may create, update, or destroy resources in particular namespaces, in both a manner aligned to operational security and developer self-service.

Resource quotas and admission control mechanisms, such as ResourceQuotas, avoid situations in which individual tenants consume too much infrastructure. Quota systems limit the number of compute units, memory, or storage in a namespace, and enforce fair sharing of resources when multiple teams or business units share the same infrastructure. To meet compliance requirements, governance layers bring data residency, encryption, and audit trail requirements to regulated workloads, as may be found in regulated industry standards [7].

VMs provide hypervisor-based isolation at the hardware level appropriate for multi-tenant environments with workloads that have different security profiles. Containers built using kernel namespaces and cgroups provide lightweight isolation, but not all container solutions are suitable for applications that need full process isolation. Research in the area of cloud security architecture has shown that layers of security mechanisms provide various types of isolation to maintain an adequate level of security [8]. Infrastructure usage, access, and configuration are tracked through audit logging and other governance capabilities. Network policy for heterogeneous compute environments provides the same firewall and traffic filtering capabilities between virtual machine and container workloads and is configured by a declarative network policy specification that makes explicit which components of applications should be able to communicate with one another.

Security Layer	Isolation Mechanism	Enforcement Method	Compliance Support
Namespace Isolation	Logical administrative domains	Kubernetes partitioning	Resource boundary separation
Role-Based Access Control	Fine-grained permissions	Per-namespace authorization	Developer self-service security
Resource Quotas	Capacity limitation	Admission control mechanisms	Fair resource sharing
VM Isolation	Hardware-level separation	Hypervisor enforcement	Multi-tenant security profiles
Container Isolation	Process-level separation	Kernel namespaces and cgroups	Lightweight workload isolation
Network Policy	Traffic filtering and segmentation	Declarative policy specifications	Cross-workload communication control
Audit Logging	Infrastructure operation tracking	Governance frameworks	Compliance reporting and forensics

Table 3: Multi-Tenancy Security and Governance Framework [7, 8]

5. Operational Patterns and Resource Optimization

The placement of workloads across many shared compute clusters requires an advanced scheduler to optimally place a workload with a given resource need inside a container or virtual machine on the cluster nodes, while maximizing the utilization of the cluster and minimizing any fragmentation. Unified scheduling techniques consider compute node capacity, memory, storage locality, virtua; hardware requirements for the workload and network topology. Scheduling policies such as affinity/anti-affinity rules, which co-locate or separate specific workloads based on whether the user wants to decrease network cost or increase availability through replica spreading in failure domains, and resource reservations are also implemented to guarantee sufficient resources to high-priority or

critical workloads. However, results from multiple unified scheduling frameworks in large-scale cluster management systems show that one single framework could handle different workloads while maximizing resource utilization and meeting application requirements through priority-based scheduling and preemption [9]. Consolidating resources removes the inefficiencies intrinsic in maintaining separate capacity pools for virtual machine and container workloads and allows dynamic scaling of resources allocated to workloads to match demand across the infrastructure estate.

By obviating the need for separate tools to manage workload deployment, monitoring, and lifecycle management operations, simplifying operations by eliminating the need to switch between tooling systems, reducing training requirements for operational staff, and providing standardized troubleshooting procedures in a heterogeneous infrastructure environment, the virtualization layer offers benefits in terms of operational simplicity. Self-service application development capabilities also benefit from standard tooling and APIs through which developers can provision infrastructure resources without knowledge of the underlying virtualization technology, which reduces development times and the dependence on a centralized infrastructure team. Monitoring and observability capabilities gather metrics, logs, and distributed tracing across all workloads for an observability view of application performance and infrastructure health.

Container orchestration research has focused on improving image and lazy loading that reduces deployment latency in environments hosting many containers [10]. Disaster recovery in a unified environment uses consistent snapshotting and replication techniques to enable the preservation of the virtual machine state and containerized application data. Deliver application acceleration via standardized processes that implement repeatable deployment pipelines of similar continuous integration and continuous deployment practices as virtual machine and container workloads to reduce time-to-production while meeting quality and compliance requirements across applications.

Operational Pattern	Optimization Strategy	Implementation Approach	Business Impact
Unified Scheduling	Resource-aware placement	Affinity/anti-affinity algorithms	Cluster utilization maximization
Workload Consolidation	Single capacity pool	Dynamic resource allocation	Infrastructure efficiency improvement
Consolidated Tooling	Single management interface	Unified deployment and monitoring	Operational complexity reduction
Developer Self-Service	Standardized API access	Infrastructure provisioning automation	Development cycle acceleration
Observability Integration	Cross-workload monitoring	Metrics, logs, distributed tracing	Performance visibility enhancement
Image Management	Efficient distribution	Lazy loading strategies	Deployment latency reduction
Disaster Recovery	Consistent protection	Snapshot and replication mechanisms	Business continuity assurance
CI/CD Standardization	Unified deployment pipelines	Repeatable workflow automation	Time-to-production optimization

Table 4: Resource Optimization and Operational Efficiency Patterns [9, 10]

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

6. AI/ML Workload Integration and Enterprise Implementation

A growing use case for unified workload management of all types is positive GPU-accelerated virtual machines. Artificial intelligence and machine learning workloads often require hardware devices unavailable to containers, like GPUs, and so require virtual machines. Model training workloads require access to large amounts of memory and persistent storage to hold the scale of data involved. These workloads can benefit from virtual machine isolation as well as Kubernetes-like orchestration. These can be important for data scientists, as they can use the same environments with known versions of the CUDA toolkit, driver, and frameworks that may be difficult to instantiate in containers without losing performance or compatibility. However, recent work in the context of GPU cluster management for distributed deep learning has identified challenges in efficiently scheduling training jobs on shared accelerator resources with respect to queueing, job fairness policies, and job completion times [11].

Containerized inference service deployment patterns leverage the lightweight and efficient characteristics of container technology to enable rapid scale-up, rollout of production deployments with versioning, and canary deployments for traffic shifting in ML model serving. Hybrid workload orchestration allows for the combination of GPU-accelerated virtual machines for computationally intensive training jobs, combined with containerized microservices for data preparation, feature engineering, and model inference with a single API. This orchestration approach allows organizations to use expensive GPU resources only during training, while also supporting cost-effective inference workloads on more standard compute infrastructure.

Enterprise case study patterns show that gradual migration is a prevailing approach, where organizations start with migrating stateless container-ready applications and then migrate missioncritical applications that are dependent on VM platforms. Cloud-native virtualization platform migration patterns follow a similar phased approach that builds unified management capabilities as a precursor to migrating workloads to reduce risk and validate operational patterns. A systematic literature review of research on cloud migration suggests that organizations should prioritize application assessment, risk assessment, and stakeholder alignment ahead of infrastructure modernization programs [12]. Additionally, techniques such as advanced service discovery and networking are required to allow legacy applications to coexist and communicate with their cloudbased counterparts. Infrastructure modernization without disruption focuses on unified control planes for managing existing virtual machine workloads in conjunction with new workloads based on containers, enabling customers to extend and not replace their existing infrastructure capabilities without having to disrupt their own digital transformation initiatives or disrupt business-critical activities.

7. Virtual Machine Provisioning and Developer Self-Service

The shift from imperative to declarative provisioning is the catalyst for Developer Self-Service. Traditional infrastructure provisioning workflows require developers to submit tickets through centralized operations teams, creating bottlenecks that can delay deployments by days or weeks. This manual intervention model fundamentally conflicts with the velocity requirements of continuous integration and continuous deployment pipelines, where infrastructure must be provisioned and decommissioned rapidly to match application lifecycle demands.

By using "VM Classes," Infrastructure Admins can pre-define "menus" of approved compute, memory, and storage profiles. These VM Classes encapsulate not only resource specifications but also organizational policies regarding security profiles, network configurations, and compliance requirements. This abstraction layer enables standardization across the organization while providing flexibility for diverse application requirements. The declarative nature of VM Classes aligns with microservices architectural patterns where infrastructure specifications are version-controlled and treated as code artifacts [5].

When a developer needs a VM, they do not open a ticket; they submit a YAML manifest specifying a VirtualMachine object that references a VirtualMachineClass. The system validates the request against the tenant's quota and automatically provisions the resource. This "API-first" approach allows infrastructure to be treated as code (IaC), enabling developers to replicate environments in development, staging, and production without manual intervention from the infrastructure team [5]. The reconciliation-based provisioning model ensures that the actual state of infrastructure converges to the desired state specified in the manifest, following the same control loop patterns established in container orchestration systems [3]. This eliminates configuration drift and enables reliable, repeatable deployments across multiple environments.

The self-service model fundamentally transforms the developer experience by removing wait states from the development workflow. Developers gain the autonomy to provision resources on demand while infrastructure teams retain governance through pre-approved VM Classes and namespace-level resource quotas. This separation of concerns enables organizations to scale their development velocity without proportionally scaling their infrastructure operations teams. Furthermore, the declarative manifest approach provides inherent documentation of infrastructure requirements and enables infrastructure changes to be reviewed through standard code review processes, improving both transparency and quality control in infrastructure management.

8. Multi-Tenancy, Governance, and the Provider-Tenant Model

In a Cloud Service Provider (CSP) or large enterprise context, the "Admin" role is not monolithic. Traditional infrastructure management models often conflate multiple administrative responsibilities, creating security risks and operational inefficiencies. Unified management empowers two distinct administrative layers that establish clear boundaries of responsibility and control, enabling both organizational governance and operational agility.

Infrastructure Admins (Providers): Manage the physical hardware, hypervisor versions, and aggregate resource pools. They define the boundaries within which tenants operate. Provider administrators are responsible for the foundational infrastructure fabric, including capacity planning, hardware lifecycle management, security patch management at the hypervisor level, and the definition of service-level agreements. This role aligns with the architectural pattern of separating the control plane from the data plane, where providers manage the underlying compute substrate while remaining abstracted from tenant-specific application concerns [3]. Provider administrators establish the VM Classes available to tenants, configure global network policies, and ensure compliance with regulatory requirements such as data residency and encryption standards.

Tenant Admins (Organizations): Operate within an assigned Namespace. They manage their own users via RBAC, distribute their allocated quota among developers, and oversee the specific applications. Tenant administrators function as project-level infrastructure managers who translate organizational requirements into operational policies within their assigned namespace boundaries. They configure role-based access control policies that determine which team members can create, modify, or delete resources, implement resource quotas to prevent individual projects from consuming disproportionate infrastructure capacity, and establish application-specific networking and security policies [7]. This organizational model enables multi-tenancy at scale by providing logical isolation between different business units or customer organizations while maintaining centralized governance and resource efficiency.

This model ensures that while developers have self-service agility, the Provider maintains central governance over data residency, encryption, and auditability [7]. The provider-tenant separation implements the principle of least privilege, where each administrative layer has precisely the permissions required for its scope of responsibility. This architectural pattern has been validated in

large-scale cluster management systems where hierarchical administrative models enable efficient resource utilization while maintaining security boundaries [9]. The governance framework ensures that tenant administrators cannot circumvent provider-level policies regarding security, compliance, or resource allocation, while simultaneously enabling tenant administrators to operate autonomously within their allocated resource boundaries without requiring provider intervention for routine operations.

The provider-tenant model also facilitates cost allocation and chargeback mechanisms, where resource consumption can be tracked and attributed to specific tenants or projects. This financial transparency enables organizations to implement showback or chargeback models that align infrastructure costs with business value, improving cost awareness and resource efficiency across the organization. Furthermore, the clear separation of responsibilities enables independent scaling of provider and tenant administrative functions, where provider capacity can be expanded to support additional tenants without increasing the operational burden on existing tenant administrators.

Persona	Responsibility	Key Abstractions
Provider	Hardware health, capacity, global security	Host Clusters, Resource Pools, VM Classes
Tenant Admin	Project-level quotas, user access, team governance	Namespaces, RBAC, ResourceQuotas
Developer	App deployment, scaling, lifecycle	VM Manifests, Pods, PVCs

Table 5: Separation of Concerns in Unified Workload Management

Conclusion

Unified workload management using Kubernetes-native APIs can deliver a step change in infrastructure operations in the enterprise. By providing a transformative new model for managing virtual machines and containers in a consolidated way through consistent declarative APIs, the architectural patterns in this guide will help you extend your Kubernetes control planes and realize operational benefits without sacrificing the isolation and compliance needs of legacy workloads. Namespace-based multi-tenancy, role-based access control, and unified scheduling further increase the motivation for a common orchestration and management layer for heterogeneous compute resources, which also include GPU-accelerated virtual machines for the training of machine learning models and containerized inference services for their deployment. Applying these patterns allows an organization to improve resource utilization, operational consistency, and developer productivity while continuing to support mission-critical workloads on virtualization platforms. It also allows a controlled incremental infrastructure

modernization approach that extends existing virtualization investments to support digital transformation initiatives. For enterprises, this provides the governance of the need to innovate against the operational imperatives. Unified workload management provides the architectural framework for the infrastructure evolution that supports cloud-native development and enterprise computing.

References

[1] David Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," IEEE Cloud Computing, Volume 1, Issue 3, 2014. [Online]. Available: https://ieeexplore.ieee.org/document/7036275

[2] Kelsey Hightower, Brendan Burns, and Joe Beda, “Kubernetes: Up and Running,” O'Reilly Media, 2017. [Online]. Available: https://www.oreilly.com/library/view/kubernetes-upand/9781491935668/

[3] Brendan Burns et al., "Borg, Omega, and Kubernetes: Lessons learned from three containermanagement systems over a decade," ACM Queue, 2016. [Online]. Available: https://queue.acm.org/detail.cfm?id=2898444

[4] Víctor Medel et al., "Characterising resource management performance in Kubernetes," Computers & Electrical Engineering, Volume 68, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0045790617315240

[5] Pooyan Jamshidi et al., "Microservices: The Journey So Far and Challenges Ahead," IEEE Software, Volume 35, Issue 3, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8354433

[6] Anna Pupykina and Giovanni Agosta, "Survey of Memory Management Techniques for HPC and Cloud Computing," IEEE Access, 2019. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8906102

[7] Michael Armbrust et al., "A view of cloud computing," Communications of the ACM, 2010. [Online]. Available: https://cacm.acm.org/practice/a-view-of-cloud-computing/

[8] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," Journal of Network and Computer Applications, Volume 34, Issue 1, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S1084804510001281

[9] 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

[10] Tyler Harter et al., "Slacker: Fast distribution with lazy Docker containers," FAST'16: Proceedings of the 14th USENIX Conference on File and Storage Technologies, 2016. [Online]. Available: https://dl.acm.org/doi/10.5555/2930583.2930597

[11] Juncheng Gu et al., "Tiresias: A GPU Cluster Manager for Distributed Deep Learning," 2019. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/gu

[12] Pooyan Jamshidi et al., "Cloud Migration Research: A Systematic Review," ResearchGate, 2014. [Online]. Available: https://www.researchgate.net/publication/260420072_Cloud_Migration_Research_A_Systematic_Review