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

Research Article

The Evolution of Operating System Paradigms in Distributed Computing

Krishna Sai Sevilimedu Veeravalli
Scadea Solutions Inc., USA

ARTICLE INFO

Received: 13 Jan 2026

Revised: 16 Jan 2026

ABSTRACT

The evolution of distributed computing infrastructure has fundamentally transformed how concurrent processes are managed across global environments, necessitating new theoretical frameworks to understand these complex systems. This article proposes the Micro-Operating System Model (μ-OSM), a conceptual framework that views cloud-scale runtimes as distributed micro-kernels governing lightweight, communicating entities. By drawing explicit parallels between traditional operating system functions and their distributed counterparts, μ-OSM offers a cohesive theoretical foundation for understanding large-scale distributed systems. The model abstracts fundamental primitives—distributed processes, schedulers, message buses, and fault cells—to enable reasoning about global concurrency as an extension of established operating system design principles. This article examines how contemporary frameworks like cluster managers and actor systems implicitly function as "meta-operating systems," highlights the current theoretical fragmentation in the field, and demonstrates how μ-OSM facilitates meaningful comparison between disparate systems through a unified vocabulary of scheduling semantics, isolation mechanisms, and recovery protocols. Future research directions, including distributed scheduling theory, global fault domain semantics, cross-platform abstraction layers, and performance modeling, are explored.

Keywords: Distributed Computing, Micro-Operating System Model, Cloud Infrastructure, Fault Tolerance, Resource Scheduling

1. Reconceptualizing Cloud Infrastructure Through an Operating System Lens

Modern distributed computing has transitioned from the concept of localized computing to one encompassing large-scale, geographically dispersed systems. This has called for fundamental rethinking regarding the design, management, scheduling, and coordination of concurrent processes across heterogeneous environments spread over several geographical regions. Such large and complex systems pose unprecedented challenges in terms of resource management, fault tolerance, and performance optimization in view of diverse hardware and network topologies [1].

It follows that, in moving beyond the boundaries of a single machine, applications necessarily face fundamental distributed system challenges, such as partial failures, network partitions, and wide variations in latency. These constraints have driven system designers to create an increasingly sophisticated orchestration layer managing the lifecycle, placement, and communication of distributed processes [2]. The resemblance between these coordination mechanisms and traditional operating system functions has grown and grown, hinting at deeper theoretical ties.

The studies of the latest research introduce a new theoretical framework-the so-called MicroOperating System Model (μ-OSM)-according to which cloud-scale runtimes are distributed microkernels. This framework is a direct adaptation of the classic operating system functions into their counterparts in distributed environments in clouds, and thus offers a consistent theoretical basis to large-scale distributed systems. By abstracting scheduling, communication, and fault tolerance into a uniform model, μ-OSM offers a systematic method to study the behavior and properties of globally distributed computing environments.

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

2. The Distributed Operating System Paradigm

The classic operating systems were intended to coordinate processes that were operating in the address space of a single machine, local scheduling, memory protection, and process-to-process communication. The migration to cloud-native designs has greatly broadened this to consider systems that coordinate operations in data centers with high latencies, which may experience partial failure, and are under different administrative domains. The challenges of distributed coordination have necessitated sophisticated schedulers that make placement decisions with imperfect information, as demonstrated in systems like Omega [3].

Unlike monolithic schedulers that maintain exclusive control over resources, shared-state schedulers described by Schwarzkopf et al. allow multiple independent components to make concurrent decisions over shared resources, employing optimistic concurrency control for conflict resolution [3]. This represents a shift toward flexible, decentralized control structures capable of handling the scale of modern cloud environments.

Communication across a distributed system inherently adds complexity over and above conventional inter-process communication. Message-passing paradigms described by Culler et al. take network latency, bandwidth limits, and topology-aware routing into consideration to achieve performance comparable to local communication [4]. Modern frameworks have implemented message buses that give delivery guarantees while handling network partitions and asynchronous delivery challenges that are not seen in single-machine environments.

Resource isolation in distributed settings extends protection domains across machine boundaries. The hardware/software co-design approach advocated by Culler emphasizes how abstraction layers must work in concert to maintain performance without sacrificing protection [4]. Container orchestration platforms implement multi-level isolation, combining process boundaries with network policies, resource quotas, and distributed access control.

While these are significant changes, cloud systems retain conceptual lineage to classical operating system principles. Therefore, Omega's two-level scheduling framework retains core operating system concepts while adapting to a distributed environment through concurrency control and conflict resolution [3]. Similarly, the various proposed communication fabrics draw on message-passing models evolved from foundational work on parallel architectures.

This has led to the development of "meta-operating systems" for distributed environments that extend traditional concepts across machine boundaries, creating abstractions that shield developers from underlying complexity while acknowledging the fundamental differences between operating systems on a single machine and those in a distributed environment [3][4].

Feature	Traditional OS	Distributed OS
Process Management	Single machine address space	Cross-datacenter coordination
Scheduling Approach	Complete resource visibility	Imperfect information, shared-state scheduling
Communication Model	Direct inter-process communication	Message buses with partition tolerance
Resource Isolation	Machine-level protection domains	Cross-machine boundaries with multilevel policies
Failure Handling	Assumes full system availability	Designed for partial failures
Control Structure	Centralized	Decentralized and flexible

Table 1: Evolution of Operating System Concepts from Single Machine to Cloud Scale [3, 4]

3. Modern Distributed Frameworks as Meta-Operating Systems

The two decades of research in cluster management and distributed coordination have produced frameworks that, in effect, work like operating systems but at a much higher layer of abstraction.

Systems such as Google Borg, Omega, and their open-source derivatives have managed to provide comprehensive resource scheduling, fault isolation mechanisms, and lifecycle management capabilities for a wide range of workloads on distributed environments. This is considered an evolution in which the experience from dealing with big data has brought up a realization that management of large-scale infrastructures indeed encompasses issues similar to those of traditional operating systems, with added twists due to scale, heterogeneity, and distribution. This evolution comes as a result of practical experience with large-scale deployments wherein the need to share resources efficiently has fuelled increasingly sophisticated forms of scheduling and management approaches 5.

The shift from statically allocated resources to dynamically orchestrated, policy-driven platforms presents a fundamental shift in the design of distributed systems. Modern cluster managers contain sophisticated resource allocation algorithms that consider multidimensional constraints, including CPU, memory, network bandwidth, and specialized hardware accelerators. These systems must balance competing goals of fairness, utilization, and application-specific demands under dynamic environmental conditions. Such heterogeneity and dynamicity features embodied in cloud environments, as characterized by Reiss et al., inherently introduce additional complexities in resource scheduling not addressed by operating system schedulers [5]. Modern schedulers must account for diverse hardware resources, variable workload, and shifting resource availability when making placement decisions-challenges outside the realm of conventional operating system schedulers.

Similarly, event-driven runtimes and actor frameworks distribute computation via lightweight concurrency units that function analogously to traditional processes or threads. The actor model, formalized by Agha, represents a theoretical basis for reasoning about concurrent computation in distributed environments [6]. By encapsulating both state and behavior within independent actors that exchange information only through asynchronous message passing, these frameworks enable the creation of scalable, resilient applications that easily accommodate distribution. Modern manifestations of the actor model flesh out these notions with supervision hierarchies, location transparency, and cluster awareness, effectively making distributed process models that are similar to the process management offered by operating systems but with semantics tailored to distributed environments.

Messaging infrastructure is another area where distributed frameworks have evolved operating system abstractions to cope with global scale. Distributed messaging systems like Apache Kafka offer extensive durability, scalability, and partition tolerance guarantees and supplement traditional interprocess communication systems. These systems have elaborate message ordering protocols, network partitioning protocols, and exactly-once delivery semantics protocols, which are beyond the capabilities of traditional operating system messaging facilities. The log-processing approach described by Kreps et al. provides a foundation for reliable data exchange in distributed environments, supporting communication patterns impractical using conventional operating system primitives [6].

These developments signal a significant convergence: operating system abstractions have transcended the physical machine boundary and now permeate the global cloud infrastructure. Contem- porary actor systems enforce fault tolerance strategies in their hierarchies of supervision, drawing inspiration from traditional operating systems but generalized for an environment in which failure is the rule rather than the exception. Similarly, the resource allocation mechanisms in cluster managers apply scheduling theory originally developed for operating systems to the distributed context while integrating additional dimensions such as network topology and data locality. This convergence is a natural evolution of systems theory; the principles that govern resource management, isolation, and communication within a single machine have been found to carry over, with appropriate extensions, to globally distributed infrastructure.

OS Component	Traditional Function	Distributed Framework Implementation	Key Benefit
Process Manager	Local execution units	Actor frameworks & eventdriven runtimes	Location transparency

Scheduler	Single-machine resource allocation	Cluster managers (Borg, Omega)	Multi-dimensional constraint handling
IPC Mechanism	Local message passing	Distributed message buses (Kafka)	Partition tolerance & durability
Fault Handler	Exception management	Supervision hierarchies	Resilience to partial failures
Resource Isolator	Memory protection	Container orchestration	Multi-tenant infrastructure

Table 2: Meta-Operating Systems: Transforming OS Concepts for Cloud Scale [5, 6]

4. The Need for Unified Theoretical Articulation

While this convergence has defined practice in recent years, the field still does not have a consistent theoretical articulation through which to describe these systems holistically. Models developed to date tend to focus on individual components in isolation—cluster schedulers, distributed message brokers, and serverless orchestration in turn—and without a common framework through which to relate them to foundational systems theory. This is most clearly seen in the gap between traditional process calculi, whose formal semantics provide a foundation for reasoning about concurrency and communication, and ad-hoc implementation patterns that have come to dominate distributed systems engineering. Though the λ-calculus and process algebras like the π-calculus provide highly regular frameworks through which to describe computation and communication, respectively, their direct application to globally distributed systems has been limited by the practical difficulties of implementing formal models at scale [7]. The Berkeley view on serverless computing describes this dichotomy, with contemporary cloud abstractions having been developed almost purely based on engineering pragmatism, rather than based on a guiding theoretical foundation [7].

Current approaches to describing distributed systems' properties rely on a set of disconnected, domain-specific languages and models. For instance, scheduling policies tend to be specified using tailored constraint languages, while isolation guarantees are defined through security-oriented frameworks and communication patterns through messaging-oriented specifications. This artificially separates different aspects of system behavior that are inherently interconnected. It is, for instance, not possible to separate the performance implications of scheduling decisions from the underlying communication patterns exchanged by the distributed processes, as current models express these independently. Similarly, the resilience properties of a distributed application are linked to both process placement and supervision strategies, yet comprehensive frameworks that allow reasoning about such relationships remain to be devised. This lack of a common theoretical foundation complicates the task of establishing correctness or performance guarantees that span multiple subsystems within a distributed environment [8].

This subdivision has pragmatic implications for system designers who find it hard to reason about end-to-end concurrency semantics, fault domains, and fairness guarantees in a geographically distributed environment. When creating distributed applications, the developers have to make their way through a wizzy mess of loosely overlapping abstractions that provide inconsistent guarantees. A system can provide strong consistency guarantees of accessing data and best-effort delivery of notifications of events, such that it can be challenging to reason on a globally consistent basis about the reliability of distributed workflows. The datacenter-as-a-computer paradigm described by Barroso et al. emphasizes the need for cohesive abstractions enabling reasoning about distributed systems as integrated entities rather than loose collections of components [8]. Without such abstractions, the cognitive load on system designers increases substantially, yielding brittle implementations and unexpected failure modes.

The lack of a generalized model inhibits the portability of resilience mechanisms and scheduling guarantees across systems. Modern distributed systems implement similar patterns for detecting failures, supervising processes, and allocating resources, but express these patterns through environment-specific interfaces and semantics. This makes it difficult to transfer both implementation techniques as well as formal reasoning between different environments. As an example, the actor

systems supervision hierarchies, the container orchestrator reconciliation loops, and the serverless platforms error-handling middlewares have different fault tolerance aspects, but with incompatible abstractions having different recovery semantics. Similarly, the varying scheduling algorithms used in these systems tend to use similar bin-packing, load balancing, and affinity-based placement heuristics, only formulated as environment-specific configuration languages [7]. A coherent theoretical framework would be able to facilitate the creation of portable, constructable abstractions that would be directly reusable across the various execution environments, and eliminate the disaggregation that now typifies the distributed systems landscape.

Challenge Area	Current State	Impact on System Design	Potential Solution Direction
Theoretical Framework	Fragmented domainspecific models	Difficulty reasoning about end-to-end behavior	Unified conceptual model (μ-OSM)
Component Integration	Isolated component descriptions	Artificial separation of interconnected concerns	Cohesive abstraction layer
Guarantee Consistency	Varying guarantees across subsystems	Cognitive burden for developers	Common guarantee vocabulary
Pattern Portability	Environment-specific implementations	Limited transferability between platforms	Portable abstraction interfaces
Failure Semantics	Incompatible recovery abstractions	Brittle implementations	Standardized fault domain definitions

Table 3: Bridging the Theoretical Gap in Distributed Systems Design [7, 8]

5. The Micro-Operating System Model (μ-OSM)

The theoretical gap is filled by the proposed Micro-Operating System Model (μ-OSM), which views each distributed runtime-container orchestration platform, event-stream processor, or actor system as a microkernel governing a constellation of lightweight, communicating entities. Guiding this work is the principle of microkernel architecture, wherein core operating system functionality is decomposed into minimal, well-defined components whose interfaces with one another are clearly defined. In distributed settings, this decomposition extends across machine boundaries, with components potentially executing on different physical nodes yet maintaining cohesive system semantics. The μOSM framework provides a unified abstraction layer that normalizes the behavior of heterogeneous distributed systems, allowing reasoning about their properties through a common conceptual lens irrespective of implementation details [7].

In this conceptual framework, a distributed process is an independently addressable, ephemeral execution context that can span containers, functions, or actors across the distributed environment. These processes encapsulate both the computational logic to be executed and the state associated with executing this logic; their lifecycle is mediated by the surrounding framework. Unlike traditional OS processes, which are bound to a machine, distributed processes in the μ-OSM model can migrate between execution environments, replicate across failure domains, and scale horizontally to adapt to load variations. This ability allows them to adapt dynamically to changing environmental conditions while maintaining consistent process semantics. As such, a paradigm representative of this approach is that of serverless computing, which seeks to treat functions as location-independent units of execution whose instantiation should be able to occur at any point in a global infrastructure [7].

Generalizing this concept to different execution models, μ-OSM introduces a common vocabulary for the definition of computational units independent of their actual implementation.

Schedulers in the μ-OSM framework represent policy-driven concurrency control across domains of failure and latency that coordinate process execution. These implement complex decision logic, considering multi-dimensional constraints involving resource availability, data locality, fault domain

distribution, and performance objectives. As compared to operating system schedulers operating within a single resource domain, distributed schedulers must cope with heterogeneous hardware capabilities, variable network conditions, and independent failure modes across the infrastructure. As explained by Jonas et al. in the context of serverless computing, the two-level scheduling paradigm separates policy decisions from the mechanisms of resource allocation, providing flexible adaptation to diverse workload characteristics [7]. Modeling schedulers in μ-OSM as first-class entities supports reasoning about fairness, efficiency, and determinism in the execution of processes in a distributed fashion.

Message buses generalize inter-process communication beyond classical sockets or queues and embed causal and temporal guarantees in a distributed context. These are the basic communication means for distributed processes to coordinate their activities, exchange data, and keep consistent state views. Unlike local mechanisms for inter-process communication, message buses deployed over a network must consider network partitions, variable latency, and possible reorderings of messages. As a theoretical background, Lamport's formal models of distributed system behavior provide a theoretical foundation for reasoning about message ordering and causality in distributed environments [9]. Practical guarantees over delivery semantics, durability, and partition tolerance complement these formal notions and extend them to their modern implementations. Finally, by embedding such guarantees into the μ-OSM framework, this model enables reasoning about distributed communication patterns under a unified lens that takes into account both theoretical properties and practical implementation constraints.

Fault cells encapsulate supervision and recovery boundaries that function in the same way as protection domains in traditional operating systems. The cells define isolation units within which failures can be confined and contained without cascading to the larger system. Hierarchical supervision models implemented in actor systems and fault domain isolation strategies employed in cloud infrastructure illustrate how this works in real life. Making fault isolation explicit as part of the model, μ-OSM provides a device for reasoning about system resilience in partial failures, an intrinsic characteristic of a distributed environment. Byzantine fault tolerance mechanisms described by Lamport et al. provide formal models for ensuring the correctness of the system even in the presence of arbitrary failure modes and inform the design of robust distributed systems [9]. In the μ-OSM framework, these are generalized to supervision strategies applicable consistently across various execution environments.

By abstracting these basic primitives, μ-OSM allows reasoning about global concurrency not only as a networking problem but also as an extension of established operating system design principles to planetary scale. This shift in perspective recognizes the profound similarities between conventional operating system functionality and its distributed equivalents, yet it provides a structured approach to addressing the additional complexities introduced by distribution. The model's emphasis on composition and well-defined interface boundaries reflects the microkernel philosophy of minimizing core components while facilitating flexible extension through well-defined protocols. This process enables the construction of modular distributed systems whereby individual components can be independently replaced or upgraded in the same way that microkernels allow operating system evolution without affecting application execution [7]. This framework enables meaningful comparison of the behaviors of disparate systems via a common vocabulary of scheduling semantics, isolation mechanisms, and recovery protocols. μ-OSM normalizes the behaviors of heterogeneous distributed runtimes under a uniform conceptual model, thereby allowing system designers to reason about trade-offs across different implementation methodologies. For instance, the scheduling guarantees of container orchestration platforms can be directly compared with those of actor frameworks, despite their distinct implementation methodologies. Similarly, the isolation properties of serverless execution environments can be compared against those of virtual machine-based infrastructure. The resulting comparative capability allows informed architectural decisions when designing distributed systems; designers can choose components based on their respective requirements without being constrained by the abstractions of particular platforms [9].

μ-OSM Component	Primary Function	Key Characteristics	Traditional OS Parallel
Distributed Process	Execution context	Location-independent, migratable, horizontally scalable	Process
Scheduler	Policy-driven concurrency control	Multi-dimensional constraints, cross-domain coordination	CPU scheduler
Message Bus	Inter-process communication	Causal/temporal guarantees, partition tolerance	IPC mechanisms
Fault Cell	Failure containment	Supervision boundaries, recovery protocols	Protection domains

Table 4: μ-OSM: Mapping Traditional OS Concepts to Distributed Environments [7, 9]

6. Future Research Directions

Most prominently, the μ-OSM framework opens several promising avenues of future research, bridging theoretical computer science with distributed systems engineering and practical cloud infrastructure design. These research directions build upon the conceptual foundations established by the framework while addressing open challenges in distributed computing at a global scale. Distributed scheduling theory is a fertile ground for theoretical investigation, considering the development of formal models that ensure fairness in resource allocation across heterogeneous, geographically dispersed environments. Current approaches to scheduling in distributed systems are usually based on heuristics and approximation algorithms that cannot provide any formal guarantee about fairness, efficiency, and convergence. By taking methodologies from operations research and economic theory and applying them to distributed scheduling problems, it is possible to develop mechanisms for allocations with provable properties of utility maximization and resource utilization. Work on economic approaches for resource allocation within computational grids by Buyya et al. lays the foundation for market-based scheduling in distributed environments where resources are allocated based on utility functions, which capture the applications' requirements and priorities [10]. Extending these models to network effects, data locality constraints, and heterogeneous hardware capabilities will enable more sophisticated scheduling frameworks that can optimize complex objective functions while providing formal guarantees about the outcome of allocation decisions. Such formal models would be particularly important in multi-tenant environments where competing workloads need to share the infrastructure resources according to well-defined policies. Global fault domain semantics arises as another critical research direction, focused on establishing rigorous definitions for failure containment and recovery strategies in multi-region deployments. The classical fault models of distributed systems are frequently based on simplifications of assumptions concerning independence of failures and recovery, which is not true in global environments that traverse administrative boundaries and geographical localities. Researchers can construct rational structures in their reasoning about system resilience in the complex fault spaces by creating more subtle taxonomies of failure modes and recovery measures. The original study of Byzantine fault tolerance by Lamport et al. discusses the theoretical constraints of distributed consensus when arbitrary failures are allowed, but it is challenging to put practical applications of the ideas to work in cloud-scale systems. Research into compositional fault models that capture the hierarchical nature of distributed infrastructure would enable finer-grained reasoning about failure propagation and containment. Such models would also inform the design of supervision strategies that maintain system integrity even in the presence of correlated failures spanning multiple layers of the infrastructure stack. Cross-platform abstraction layers are a pragmatic research direction to define unified interfaces that regularize the behavior of various distributed runtimes under the common μOSM model. Contemporary distributed systems implement similar patterns using platform-specific interfaces and semantics, with resulting fragmentation and limited portability. Researchers could enable interoperability between diverse execution environments by developing standardized abstractions that encapsulate common distributed system patterns-process lifecycle management,

message delivery guarantees, failure detection, and recovery. The service mesh paradigm described by Morgan in the context of cloud-native applications provides a pattern for abstracting communication infrastructure from application logic, but similar approaches could be extended to other aspects of distributed system behavior [10].

Domain-specific languages for describing the requirements of distributed applications independently of specific implementation platforms would enable greater portability with preservation of the performance advantages of platform-specific optimizations. Such abstraction layers would enable the composition of distributed applications from components that may execute in distinct runtime environments, facilitating more flexible architectural patterns. Performance modeling is an important research direction in using operating system performance analysis techniques to predict and optimize the behavior of distributed systems. Traditional performance modeling approaches for single-machine systems build analytic models of system behavior, using queueing theory, stochastic processes, and empirical measurement to characterize system behavior under a range of workloads. Extending these techniques to distributed environments requires consideration of additional complexities arising from network effects, resource heterogeneity, and parallel execution. An example paper on cloud performance isolation by Delimitrou and Kozyrakis shows that statistical machine learning techniques can be applied to model the performance characteristics of co-located workloads, thus allowing for resource allocation and interference mitigation with increased efficiency [11]. Research into compositional performance models that capture the interaction between distributed components would allow end-to-end performance prediction for complex distributed applications. In turn, these models provide optimization strategies at both design time and runtime, enabling systems to adapt their configuration in response to changing environmental conditions. These research directions together address the theoretical foundation, practical implementation, and performance implications of the μ-OSM framework. In this way, researchers can also create a more holistic understanding of distributed systems that better connects theoretical models with practical engineering considerations. The integration of formal methods, empirical measurement, and systems design would enable the development of distributed systems with stronger guarantees with respect to correctness, performance, and resilience. This view of large-scale distributed systems through the operating system lens provides a principled vocabulary to researchers and practitioners for reasoning about scalability, determinism, and resilience in modern cloud environments. The framework μ-OSM serves as a conceptual bridge between traditional operating system theory and distributed systems practice, in turn enabling transfer across domains of established principles and methodologies. Instead, these design aspects lead towards a more systematic approach to the design of distributed systems, with architectural decisions based on a coherent theoretical framework rather than ad-hoc engineering solutions [11]. The μ-OSM framework represents a significant step toward unifying the increasingly fragmented landscape of distributed computing paradigms under a coherent theoretical framework. By establishing common abstractions in regard to basic concepts in distributed systems processes, schedulers, communication channels, and fault, the framework supplies meaningful comparisons between different implementation approaches. This comparison capability further enables more effectively informed architectural decisions in the design of a distributed system since various trade-offs between different platforms can be weighed within a consistent conceptual framework.

Conclusion

The Micro-Operating System Model represents a strong basis on which the evolution of distributed computing infrastructure can be understood. By explicitly recognizing parallels between traditional operating system functions and their most recent, cloud-scale incarnations, this framework serves as an underpinning for more systematic activities in designing, analyzing, and optimizing a distributed system. As computing presses onward in its inexorable march to even more distributed architectures, frameworks such as μ-OSM will become indispensable with respect to maintaining conceptual clarity and engineering rigor in the face of burgeoning complexity. It is in this context that the operating system

paradigm, re-imagined for the cloud era, provides more than an analytical instrument but a roadmap to the next generation of distributed computing platforms.

References

[1] Abhishek Verma et al., "Large-scale cluster management at Google with Borg," EuroSys '15: Proceedings of the Tenth European Conference on Computer Systems, 2015. [Online]. Available: https://dl.acm.org/doi/10.1145/2741948.2741964

[2] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan, "The Datacenter as a Computer," Springer. [Online]. Available: https://link.springer.com/book/10.1007/978-3-031-017612

[3] Malte Schwarzkopf et al., "Omega: flexible, scalable schedulers for large compute clusters," EuroSys '13: Proceedings of the 8th ACM European Conference on Computer Systems, 2013. [Online]. Available: https://dl.acm.org/doi/10.1145/2465351.2465386

[4] David Culler et al., "Parallel Computer Architecture: A Hardware/Software Approach,". [Online]. Available: https://www.iqytechnicalcollege.com/Parallel%20Computer%20Architecture.pdf

[5] Charles Reiss et al., "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," SoCC '12: Proceedings of the Third ACM Symposium on Cloud Computing, 2012. [Online]. Available: https://dl.acm.org/doi/10.1145/2391229.2391236

[6] Jay Kreps et al., "Kafka: a Distributed Messaging System for Log Processing," 2011. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf

[7] Eric Jona et al., "Cloud Programming Simplified: A Berkeley View on Serverless Computing," 2019. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf

[8] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan, "The Datacenter as a Computer," Springer. [Online]. Available: https://link.springer.com/book/10.1007/978-3-031-017612

[9] Leslie Lamport et al., "The Byzantine Generals Problem," ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 4, Issue 3, 1982. [Online]. Available: https://dl.acm.org/doi/10.1145/357172.357176

[10] Rajkumar Buyya, David Abramson, and Jonathan Giddy, "An Economy Driven Resource Management Architecture for Global Computational Power Grids,". [Online]. Available: http://www.buyya.com/papers/GridEconomy.pdf

[11] Christina Delimitrou and Christos Kozyrakis, "Quasar: resource-efficient and QoS-aware cluster management," ASPLOS '14: Proceedings of the 19th international conference on Architectural support for programming languages and operating systems, 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2541940.2541941