Stout in Computer Science and Technology Studies (SCSTS)

DOI: https://doi.org/10.61424/6vvbmj66
Journal Homepage: https://stoutjournals.org/index.php/SCSTS

Containerized Application Deployment on Microsoft Azure Kubernetes Service (AKS)

Shekar Rao Lakavath
Independent Researcher, Bonneylake, Washington, USA
Corresponding Author: Shekar Rao Lakavath, **E-mail**: Shekar.lakavath@gmail.com

ARTICLE INFORMATION

Submitted: May 20, 2025
Accepted: August 15, 2025
Published: December 29, 2025
Volume: 1
Issue: 1

KEYWORDS

Containerization, Kubernetes, Microsoft Azure, Azure Kubernetes Service, DevOps, microservices, CI/CD, cloud-native computing.

ABSTRACT

Containerization has become the dominant packaging model for modern cloud-native applications, and Kubernetes has emerged as the de facto standard for orchestrating containers at scale. Microsoft Azure Kubernetes Service (AKS) extends this orchestration capability as a managed platform, removing much of the operational burden associated with running a self-managed Kubernetes control plane while retaining full compatibility with the Kubernetes application programming interface and ecosystem. This article examines how containerized applications can be reliably built, deployed, scaled, and monitored on AKS, situating the discussion within the broader evolution of container orchestration research from early cluster schedulers such as Google Borg, Apache Mesos, and Omega through to Kubernetes itself. The article traces the technical lineage of container orchestration, compares AKS against Azure Container Instances and self-managed Kubernetes deployments, and proposes an end-to-end continuous integration and continuous delivery (CI/CD) pipeline architecture spanning source control, automated build and test, container image management through Azure Container Registry, cluster deployment, and post-deployment observability. Two tables and one pipeline architecture figure illustrate the comparative platform analysis and the mapping between common deployment challenges and AKS-specific mitigations. The findings indicate that AKS meaningfully reduces the operational overhead documented in the container orchestration literature, particularly around control-plane management, security patching, and autoscaling, while requiring organizations to still invest deliberately in image security scanning, deployment automation discipline, and cluster observability to realize these benefits in production. The article concludes with practical recommendations for teams adopting AKS for containerized application deployment.

1. Introduction

The shift from monolithic application architectures toward loosely coupled microservices has been accompanied by a parallel shift in how applications are packaged and deployed. Containers, which bundle an application together with its runtime dependencies into a single portable unit, address the long-standing problem of inconsistent behavior across development, testing, and production environments (Bernstein, 2014). As organizations moved from running a handful of containers on individual hosts to running thousands of containers across large clusters, the need for automated orchestration, covering scheduling, scaling, networking, and failure recovery, became apparent, giving rise to systems such as Google's Borg and Omega, Apache Mesos, and ultimately Kubernetes (Burns et al., 2016; Schwarzkopf, Konwinski, Abd-El-Malek, & Wilkes, 2013).

Kubernetes has since become the dominant open-source container orchestration platform, and major cloud providers now offer managed Kubernetes services that remove much of the operational complexity of running a production-grade cluster. Microsoft Azure Kubernetes Service (AKS) is one such offering, providing a managed control plane, integrated identity management through Azure Active Directory, and native connectivity to the broader Azure ecosystem, including Azure Container Registry, Azure Monitor, and Azure DevOps or GitHub Actions for continuous integration and delivery.

Copyright: © 2025 The Author(s). This open access article is distributed under the terms and conditions of the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC- BY-NC-SA) License (https://creativecommons.org/licenses/by-nc-sa/4.0/), published by Stout Journals, London, United Kingdom

Despite the operational simplifications that managed Kubernetes services provide, deploying containerized applications reliably at scale remains a nontrivial engineering problem. Organizations must still design an appropriate CI/CD pipeline, configure autoscaling correctly, secure container images and cluster networking, and establish sufficient observability to detect and diagnose failures after deployment. This article addresses three questions. First, how has container orchestration technology evolved to the point of managed platforms such as AKS, and what problems does each stage of that evolution solve? Second, how does AKS compare with alternative Azure deployment models for containerized workloads? Third, what are the most common technical challenges in deploying containerized applications on AKS, and how does the platform, combined with sound engineering practice, address them? The remainder of the article reviews the relevant literature, proposes a CI/CD pipeline architecture for AKS, compares deployment options, maps deployment challenges to mitigations, and discusses practical implications for engineering teams.

2. Literature Review

2.1 From Virtual Machines to Containers

Container-based virtualization emerged as a lighter-weight alternative to hypervisor-based virtual machines, sharing the host operating system kernel while isolating application processes in separate namespaces. Bernstein (2014) provided one of the earliest widely cited overviews of this shift, describing how Docker standardized the packaging and distribution of containers and how Kubernetes subsequently emerged as a cluster manager for coordinating large numbers of containers. Felter, Ferreira, Rajamony, and Rubio (2015) conducted a rigorous empirical performance comparison of virtual machines and Linux containers, finding that containers consistently achieved performance closer to bare-metal execution, particularly for central-processing-unit- and memory-intensive workloads, while incurring negligible overhead relative to hypervisor-based virtualization. Kozhirbayev and Sinnott (2017) extended this comparison across multiple container-based technologies for cloud environments and reached similar conclusions, reinforcing container adoption as a performance-neutral or performance-positive packaging decision for most enterprise workloads.

2.2 The Evolution of Cluster Schedulers and Orchestrators

Before Kubernetes achieved widespread adoption, several large-scale cluster management systems established the core concepts that orchestration platforms would later formalize. Hindman et al. (2011) introduced Mesos, a two-level scheduler that allowed multiple frameworks to share a single cluster's resources through fine-grained resource offers. Vavilapalli et al. (2013) proposed Apache Hadoop YARN as a resource negotiator that separated cluster resource management from application scheduling logic, a separation of concerns that later orchestration systems, including Kubernetes, would also adopt. Schwarzkopf, Konwinski, Abd-El-Malek, and Wilkes (2013) described Omega, a scheduler developed at Google that used shared-state optimistic concurrency control to coordinate scheduling decisions across multiple independent schedulers operating on the same cluster.

Burns, Grant, Oppenheimer, Brewer, and Wilkes (2016) subsequently documented the lessons learned from a decade of operating Google's Borg and Omega systems, arguing that these lessons directly informed the design of Kubernetes, particularly its declarative configuration model, its use of a reconciliation control loop to continuously drive cluster state toward a desired specification, and its pod abstraction for co-scheduling tightly coupled containers. Verma et al. (2015) provided a detailed empirical account of Borg's scheduling and resource utilization characteristics at Google's operational scale, offering a benchmark against which later Kubernetes-based systems could be evaluated.

2.3 Kubernetes-Specific Research and Container Orchestration Surveys

As Kubernetes matured, researchers began publishing detailed technical analyses of its performance and resource management behavior. Medel, Rana, Bañares, and Arronategui (2016) modeled Kubernetes' performance and resource management mechanisms, providing an early quantitative basis for understanding how scheduling decisions and resource requests interact with cluster-level performance. Casalicchio (2019) surveyed the container orchestration landscape more broadly, classifying orchestration platforms according to their scheduling algorithms, scaling mechanisms, and failure recovery strategies, and identifying Kubernetes' extensible controller architecture as a key differentiator relative to earlier systems. Casalicchio and Iannucci (2020) extended this work with a state-of-the-art review of container technologies spanning application packaging, orchestration, and security, concluding that security tooling had lagged behind orchestration functionality in most container ecosystems.

Rodriguez and Buyya (2019) proposed a taxonomy of container-based cluster orchestration systems, organizing prior research according to scheduling policy, elasticity mechanism, and fault-tolerance approach, and identifying open research challenges in multi-cluster and multi-cloud orchestration that remain relevant to organizations deploying across hybrid Azure and on-premises environments. Pahl, Brogi, Soldani, and Jamshidi (2019) conducted a systematic review of cloud container technologies, concluding that container orchestration research had matured from foundational scheduling problems toward higher-level

concerns such as service mesh integration, multi-tenancy isolation, and declarative application lifecycle management, all of which are directly relevant to production AKS deployments.

2.4 Microservices, DevOps, and Deployment Automation

The adoption of containers has been closely linked to the adoption of microservices architectures and DevOps practices. Balalaie, Heydarnoori, and Jamshidi (2016) documented a case study of migrating a monolithic application to a microservices architecture, finding that containerization and orchestration were essential enablers of the continuous deployment practices that DevOps methodologies require. Kang, Le, and Tao (2016) proposed a container and microservice-driven design pattern for cloud infrastructure DevOps, arguing that treating infrastructure configuration as versioned, declarative code, an approach directly supported by Kubernetes manifests and Helm charts, substantially reduces deployment errors relative to manual configuration processes.

Amaral et al. (2015) empirically evaluated microservices architectures deployed using containers, finding that inter-service communication overhead and container placement decisions materially affected end-to-end application latency, a finding with direct implications for how AKS node pools and pod affinity rules should be configured for latency-sensitive workloads. Abdollahi Vayghan, Saied, Toeroe, and Khendek (2018) conducted controlled experiments deploying microservice-based applications on Kubernetes, documenting specific lessons learned regarding rolling update strategies, readiness and liveness probe configuration, and the operational tradeoffs of different deployment strategies, including blue-green and canary deployments.

Kratzke and Quint (2017) conducted a systematic mapping study of cloud-native applications a decade into the cloud computing era, concluding that container orchestration platforms had become the primary architectural substrate for cloud-native design patterns, including circuit breakers, service discovery, and externalized configuration, all of which are natively supported within the Kubernetes ecosystem that AKS provides.

2.5 Container and Cluster Security

Security has remained a persistent concern throughout the evolution of container orchestration. Combe, Martin, and Di Pietro (2016) examined Docker from a security perspective, identifying container image provenance, kernel-sharing risks, and privilege escalation as the primary attack surfaces requiring mitigation. Sultan, Ahmad, and Dimitriou (2019) conducted a broader survey of container security issues and challenges, categorizing risks across the image build process, the container runtime, and the orchestration layer, and recommending a defense-in-depth approach combining image scanning, runtime monitoring, and network segmentation, an approach that maps closely onto the combination of Azure Container Registry scanning and Kubernetes network policies available within AKS.

Naha et al. (2018) situated container security within the broader context of fog and edge computing architectures, noting that as containerized workloads move closer to the network edge, the attack surface expands to include less centrally managed infrastructure, a consideration relevant to AKS deployments that extend to Azure IoT Edge or Azure Arc-enabled clusters. Kaiser, Haq, Tosun, and Korkmaz (2022) extended container security and performance analysis to ARM-based architectures, reflecting the growing diversity of compute architectures that managed Kubernetes platforms must now support.

2.6 Scaling, Cluster Management, and Operational Tooling

A final stream of research has focused on the operational tooling required to run container clusters reliably at scale. Al-Dhuraibi, Paraiso, Djarallah, and Merle (2017) proposed an autonomic vertical elasticity mechanism for Docker containers that dynamically adjusts resource allocations in response to observed load, conceptually related to the Vertical Pod Autoscaler available within AKS. Casalicchio and Perciballi (2017) compared relative and absolute resource utilization metrics for container autoscaling decisions and found that relative metrics produced more stable scaling behavior under variable load, a finding directly applicable to configuring the Horizontal Pod Autoscaler in production AKS clusters. Peinl, Holzschuher, and Pfitzer (2016) surveyed Docker cluster management tools and concluded that managed orchestration platforms would likely surpass self-managed tools as the dominant deployment model, a prediction borne out by the subsequent adoption of services such as AKS.

Taken together, this literature traces a clear technical progression: from foundational cluster schedulers, through container-specific orchestration research, to the microservices and DevOps practices that containers enable, and finally to the security and operational tooling required to run containerized workloads safely in production. The following sections apply these findings to the specific context of deploying containerized applications on Azure Kubernetes Service.

3. A CI/CD Pipeline Architecture for Containerized Deployment on AKS

Building on the DevOps and deployment automation research reviewed above (Balalaie, Heydarnoori, & Jamshidi, 2016; Kang, Le, & Tao, 2016), this article proposes an end-to-end pipeline architecture for deploying containerized applications on AKS. Figure 1 illustrates the pipeline, which spans source control, automated build and test, container registry management, cluster deployment, and post-deployment observability.

Figure 1. CI/CD Pipeline for Containerized Application Deployment on Azure Kubernetes Service (AKS)

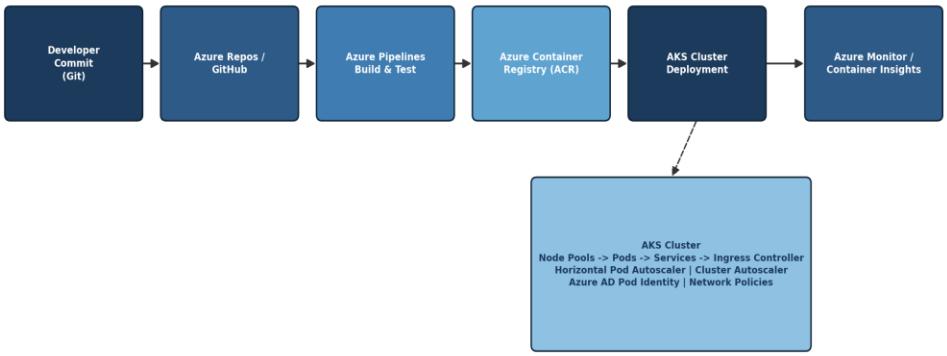


Figure 1. CI/CD pipeline for containerized application deployment on Azure Kubernetes Service (AKS), showing the flow from developer commit through build, registry, cluster deployment, and monitoring, with the internal AKS cluster components detailed below the deployment stage.

3.1 Source Control and Build Automation

Developer commits to Azure Repos or GitHub trigger Azure Pipelines, which builds the application, executes automated tests, and produces a container image. Treating the resulting image as immutable once built, and promoting the identical image artifact through development, staging, and production environments, directly addresses the environment-consistency problem identified by Bernstein (2014) as one of containerization's primary benefits.

3.2 Container Registry Management

Built images are pushed to Azure Container Registry (ACR), which performs automated vulnerability scanning before images are made available for deployment, addressing the image-provenance risks identified by Combe, Martin, and Di Pietro (2016). ACR's geo-replication capability also allows images to be pulled with low latency by AKS clusters deployed across multiple Azure regions.

3.3 AKS Cluster Deployment

Deployment to the AKS cluster is performed declaratively, using Kubernetes manifests or Helm charts that specify the desired state of Deployments, Services, and Ingress resources. Within the cluster, the Horizontal Pod Autoscaler adjusts the number of running pod replicas based on observed resource utilization, following the relative-metric autoscaling approach validated by Casalicchio and Perciballi (2017), while the Cluster Autoscaler adjusts the number of underlying virtual machine nodes in response to aggregate scheduling pressure. Azure AD Pod Identity allows individual pods to authenticate to other Azure services using managed identities rather than embedded credentials, and Kubernetes network policies restrict pod-to-pod traffic according to the principle of least privilege recommended by Sultan, Ahmad, and Dimitriou (2019).

3.4 Observability

Azure Monitor and Container Insights collect logs, metrics, and traces from both the AKS control plane and running workloads, providing the operational visibility that Casalicchio and Iannucci (2020) identify as a persistent gap in many container deployments. Alerts configured on these metrics allow operations teams to detect degraded performance or failed deployments before they affect end users.

4. Comparative Analysis of Azure Containerized Deployment Options

Organizations deploying containerized workloads on Azure can choose among several deployment models with different tradeoffs between operational control and management overhead. Table 1 compares Azure Kubernetes Service against Azure

Container Instances and a self-managed Kubernetes deployment on Azure virtual machines, across the dimensions most relevant to production container deployment.

Table 1. Comparison of Azure Containerized Application Deployment Options

Feature	Azure Kubernetes Service (AKS)	Azure Container Instances (ACI)	Self-Managed Kubernetes on Azure VMs
Orchestration	Full Kubernetes API, managed control plane	None (single container / container group)	Full Kubernetes API, self-managed control plane
Scaling	Horizontal Pod Autoscaler and Cluster Autoscaler, node pool scaling	Manual per-container scaling	Manual configuration of autoscaling components
Operational Overhead	Low; Microsoft manages control plane, patching, and upgrades	Very low; no cluster to manage	High; team manages control plane, etcd, upgrades, and patching
Networking	Azure CNI or kubenet, network policies, ingress controllers	Basic virtual network integration	Fully customizable, requires manual configuration
Identity & Access	Azure AD integration, Azure RBAC, pod-managed identities	Azure AD for resource access only	Manual integration required
Best Suited For	Production microservices requiring orchestration, scaling, and resilience	Short-lived or bursty single-container workloads	Organizations requiring full control over cluster configuration

As Table 1 indicates, AKS occupies a middle position between the minimal orchestration of Azure Container Instances and the full operational responsibility of a self-managed cluster, consistent with the broader managed-Kubernetes trend anticipated by Peinl, Holzschuher, and Pfitzer (2016). For most production microservices workloads requiring autoscaling, rolling updates, and multi-service coordination, AKS provides the orchestration capability of a full Kubernetes deployment without the control-plane operational burden documented by Verma et al. (2015) in the context of large-scale, self-managed cluster schedulers.

5. Mapping Deployment Challenges to AKS-Based Mitigations

Table 2 synthesizes the containerized deployment challenges most frequently documented in the literature reviewed above and maps each to the corresponding mitigation available within an AKS-based deployment architecture, along with representative supporting sources.

Table 2. Containerized Deployment Challenges and AKS-Based Mitigations

Containerized Deployment Challenge	AKS-Based Mitigation	Supporting Literature
Inconsistent behavior between development, test, and production environments	Immutable container images built once in Azure Pipelines and promoted unchanged through environments	Bernstein (2014); Kratzke & Quint (2017)
Manual, error-prone scaling under variable load	Horizontal Pod Autoscaler and Cluster Autoscaler adjust pod and node counts automatically based on metrics	Medel, Rana, Bañares, & Arronategui (2016); Al-Dhuraibi, Paraiso, Djarallah, & Merle (2017)
Complex cluster control-plane operations (upgrades, patching, etcd management)	AKS provides a managed, free control plane maintained by Microsoft	Verma et al. (2015); Rodriguez & Buyya (2019)
Container and image-level security vulnerabilities	Azure Container Registry vulnerability scanning combined with Kubernetes network policies and pod security standards	Combe, Martin, & Di Pietro (2016); Sultan, Ahmad, & Dimitriou (2019)
Difficulty coordinating microservice deployments and rollbacks	Kubernetes Deployments and Helm charts support declarative, versioned, and reversible rollouts	Balalaie, Heydarnoori, & Jamshidi (2016); Abdollahi Vayghan, Saied, Toeroe, & Khendek (2018)
Limited visibility into application and cluster health after deployment	Azure Monitor and Container Insights provide centralized logging, metrics, and alerting across the cluster	Casalicchio & Iannucci (2020)
Performance overhead relative to bare-metal or virtual machine deployment	Container-based virtualization on AKS nodes achieves near-bare-metal performance for most workload classes	Felter, Ferreira, Rajamony, & Rubio (2015); Kozhirbayev & Sinnott (2017)

The mapping in Table 2 illustrates that AKS mitigates many, but not all, of the challenges documented in the container orchestration literature automatically. Autoscaling and control-plane management are substantially simplified by the managed platform, but image security scanning, network policy configuration, and application-level observability still require deliberate engineering effort, consistent with Casalicchio and Iannucci's (2020) observation that container security tooling has generally lagged behind orchestration functionality across the industry.

6. Discussion

The analysis presented above supports several practical conclusions for engineering teams adopting AKS for containerized application deployment. First, the performance case for containerization over virtual-machine-based deployment is well established empirically; Felter, Ferreira, Rajamony, and Rubio (2015) and Kozhirbayev and Sinnott (2017) both found that containers achieve near-bare-metal performance for the majority of workload classes, meaning that the primary justification for choosing AKS over alternative deployment models should rest on orchestration and operational requirements rather than raw performance concerns.

Second, the reduction in control-plane operational burden that AKS provides is substantial but not absolute. While Microsoft manages the Kubernetes control plane itself, including its availability and patching, organizations remain responsible for node pool configuration, workload-level autoscaling policy, network policy design, and image security, echoing the shared-responsibility pattern documented across the broader cloud security literature. Teams that treat AKS adoption purely as an infrastructure simplification, without corresponding investment in these workload-level configurations, are likely to encounter the same categories of deployment challenges documented in self-managed Kubernetes research (Rodriguez & Buyya, 2019).

Third, the CI/CD pipeline design proposed in this article treats the container image as the unit of promotion between environments, consistent with the environment-consistency argument that has motivated container adoption since Bernstein's

(2014) foundational overview. Automating this promotion process, including automated testing and vulnerability scanning at the registry stage, is essential; manual or ad hoc deployment processes reintroduce exactly the configuration drift and human-error risks that containerization and orchestration are designed to eliminate (Kang, Le, & Tao, 2016).

Fourth, autoscaling configuration requires careful attention to the metrics used to trigger scaling decisions, since Casalicchio and Perciballi (2017) found that relative resource utilization metrics produce more stable autoscaling behavior than absolute metrics, directly informing how the Horizontal Pod Autoscaler should be tuned in production AKS clusters. Finally, security remains the area requiring the most deliberate, ongoing engineering attention: Sultan, Ahmad, and Dimitriou's (2019) defense-in-depth recommendation, combining image scanning, runtime monitoring, and network segmentation, maps directly onto capabilities available within AKS and Azure Container Registry, but these capabilities must be actively configured and maintained rather than assumed to be enabled by default.

7. Recommendations

Organizations deploying containerized applications on AKS should: (1) treat container images as immutable build artifacts, promoted unchanged from development through production via an automated Azure Pipelines or GitHub Actions workflow; (2) enable Azure Container Registry vulnerability scanning and block deployment of images with unresolved critical findings; (3) configure the Horizontal Pod Autoscaler using relative resource utilization metrics, following the stability findings of Casalicchio and Perciballi (2017), and validate scaling behavior under realistic load before production release; (4) apply Kubernetes network policies to enforce least-privilege pod-to-pod communication and use Azure AD Pod Identity rather than embedded credentials for pods that access other Azure services; (5) deploy Azure Monitor and Container Insights from the outset of any production workload, rather than adding observability retroactively after an incident; and (6) adopt declarative deployment strategies, such as rolling updates, blue-green, or canary deployments, using Kubernetes-native tooling or Helm, to minimize the operational risk of each release, following the lessons documented by Abdollahi Vayghan, Saied, Toeroe, and Khendek (2018).

8. Conclusion

Container orchestration has evolved considerably from the early cluster schedulers developed at Google and in the open-source Mesos project to the mature, managed Kubernetes platforms available from major cloud providers today. Microsoft Azure Kubernetes Service substantially reduces the operational burden of running a production Kubernetes cluster by managing the control plane and providing native integration with Azure's identity, container registry, and monitoring services. This article has proposed a CI/CD pipeline architecture for containerized application deployment on AKS, compared AKS against alternative Azure deployment models, and mapped common container deployment challenges to the specific mitigations AKS provides. The literature reviewed indicates that while AKS meaningfully simplifies control-plane operations and autoscaling, organizations must still invest deliberately in image security scanning, network policy configuration, and observability to fully realize the reliability and security benefits that container orchestration research has established as achievable. Future work should empirically evaluate production AKS deployments across diverse workload types to quantify the operational and cost benefits of managed Kubernetes relative to self-managed alternatives under real-world conditions.

Funding: This research received no external funding.
Conflicts of Interest: The authors declare no conflict of interest.
Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers
Artificial Intelligence (AI) Use Disclosure: The authors declare that no artificial intelligence tools were used in the preparation of this manuscript.

References

[1] Abdollahi V, L., Saied, M. A., Toeroe, M., & Khendek, F. (2018). Deploying microservice based applications with Kubernetes: Experiments and lessons learned. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD) (pp. 970–973). IEEE. https://doi.org/10.1109/CLOUD.2018.00148

[2] Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., & Merle, P. (2017). Autonomic vertical elasticity of Docker containers with ElasticDocker. In 2017 IEEE 10th International Conference on Cloud Computing (CLOUD) (pp. 472–479). IEEE. https://doi.org/10.1109/CLOUD.2017.67

[3] Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., & Steinder, M. (2015). Performance evaluation of microservices architectures using containers. In 2015 IEEE 14th International Symposium on Network Computing and Applications (NCA) (pp. 27–34). IEEE. https://doi.org/10.1109/NCA.2015.49

[4] Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices architecture enables DevOps: Migration to a cloud-native architecture. IEEE Software, 33(3), 42–52. https://doi.org/10.1109/MS.2016.64

[5] Bernstein, D. (2014). Containers and cloud: From LXC to Docker to Kubernetes. IEEE Cloud Computing, 1(3), 81–84. https://doi.org/10.1109/MCC.2014.51

[6] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. Communications of the ACM, 59(5), 50–57. https://doi.org/10.1145/2890784

[7] Casalicchio, E. (2019). Container orchestration: A survey. In A. Puliafito & K. S. Trivedi (Eds.), Systems Modeling: Methodologies and Tools (pp. 221–235). Springer. https://doi.org/10.1007/978-3-319-92378-9_14

[8] Casalicchio, E., & Iannucci, S. (2020). The state-of-the-art in container technologies: Application, orchestration and security. Concurrency and Computation: Practice and Experience, 32(17), e5668. https://doi.org/10.1002/cpe.5668

[9] Casalicchio, E., & Perciballi, V. (2017). Auto-scaling of containers: The impact of relative and absolute metrics. In 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W) (pp. 207–214). IEEE. https://doi.org/10.1109/FAS-W.2017.149

[10] Combe, T., Martin, A., & Di Pietro, R. (2016). To Docker or not to Docker: A security perspective. IEEE Cloud Computing, 3(5), 54–62. https://doi.org/10.1109/MCC.2016.100

[11] Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015). An updated performance comparison of virtual machines and Linux containers. In 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (pp. 171–172). IEEE. https://doi.org/10.1109/ISPASS.2015.7095802

[12] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., & Stoica, I. (2011). Mesos: A platform for fine-grained resource sharing in the data center. In Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI) (pp. 295–308). USENIX Association.

[13] Kaiser, S., Haq, M. S., Tosun, A., & Korkmaz, T. (2022). Container technologies for ARM architecture: A comprehensive survey of the state-of-the-art. IEEE Access, 10, 84853–84881. https://doi.org/10.1109/ACCESS.2022.3197126

[14] Kang, H., Le, M., & Tao, S. (2016). Container and microservice driven design for cloud infrastructure DevOps. In 2016 IEEE International Conference on Cloud Engineering (IC2E) (pp. 202–211). IEEE. https://doi.org/10.1109/IC2E.2016.26

[15] Kozhirbayev, Z., & Sinnott, R. O. (2017). A performance comparison of container-based technologies for the cloud. Future Generation Computer Systems, 68, 175–182. https://doi.org/10.1016/j.future.2016.08.025

[16] Kratzke, N., & Quint, P. C. (2017). Understanding cloud-native applications after 10 years of cloud computing: A systematic mapping study. *Journal of Systems and Software*, 126, 1–16. https://doi.org/10.1016/j.jss.2017.01.001

[17] Medel, V., Rana, O., Bañares, J. Á., & Arronategui, U. (2016). Modelling performance & resource management in Kubernetes. In 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC) (pp. 257–262). IEEE/ACM. https://doi.org/10.1145/2996890.3007869

[18] Naha, R. K., Garg, S., Georgakopoulos, D., Jayaraman, P. P., Gao, L., Xiang, Y., & Ranjan, R. (2018). Fog computing: Survey of trends, architectures, requirements, and research directions. IEEE Access, 6, 47980–48009. https://doi.org/10.1109/ACCESS.2018.2866491

[19] Pahl, C., Brogi, A., Soldani, J., & Jamshidi, P. (2019). Cloud container technologies: A state-of-the-art review. IEEE Transactions on Cloud Computing, 7(3), 677–692. https://doi.org/10.1109/TCC.2017.2702586

[20] Peinl, R., Holzschuher, F., & Pfitzer, F. (2016). Docker cluster management for the cloud: Survey results and own solution. *Journal of Grid Computing*, 14(2), 265–282. https://doi.org/10.1007/s10723-016-9366-y

[21] Rodriguez, M. A., & Buyya, R. (2019). Container-based cluster orchestration systems: A taxonomy and future directions. Software: Practice and Experience, 49(5), 698–719. https://doi.org/10.1002/spe.2660

[22] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., & Wilkes, J. (2013). Omega: Flexible, scalable schedulers for large compute clusters. In Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys) (pp. 351–364). ACM. https://doi.org/10.1145/2465351.2465386

[23] Sultan, S., Ahmad, I., & Dimitriou, T. (2019). Container security: Issues, challenges, and the road ahead. IEEE Access, 7, 52976–52996. https://doi.org/10.1109/ACCESS.2019.2911732

[24] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., & Baldeschwieler, E. (2013). Apache Hadoop YARN: Yet another resource negotiator. In Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC) (Article 5). ACM. https://doi.org/10.1145/2523616.2523633

[25] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Large-scale cluster management at Google with Borg. In Proceedings of the 10th European Conference on Computer Systems (EuroSys) (Article 18). ACM. https://doi.org/10.1145/2741948.2741964