RESEARCH ARTICLE

Container Orchestration in Microservices: Kubernetes vs. Docker Swarm

Neda Abdelhamid
Computing and Informatics Department, De Montfort University, Leicester, UK

Received: 25/01/2018 **Accepted:** 13/02/2018 **Published:** 30/03/2018

Abstract

Container orchestration platforms have become vital tools for deploying and managing microservices-based applications in production environments. This paper presents a comparative analysis of Kubernetes and Docker Swarm, focusing on deployment efficiency, scalability, fault tolerance, and ease of management. Using a sample e-commerce microservices application, we evaluate both systems under varying loads, node failures, and configuration complexities. Kubernetes demonstrates robust auto-scaling, dynamic scheduling, and self-healing capabilities, offering better support for stateful applications and resource quotas. Its declarative configuration model and broad ecosystem make it suitable for complex, multi-service applications. Docker Swarm, while more lightweight, provides faster startup times and a simpler learning curve, which is advantageous for smaller teams or limited-resource environments. Benchmarks show Kubernetes handles larger service graphs with higher stability, while Swarm's performance advantage diminishes as service count and cluster size grow. We also discuss network configurations, persistent storage management, and CI/CD integration. Our findings suggest that the choice between the two depends on organizational needs—Kubernetes excels in feature-rich, large-scale environments, whereas Docker Swarm offers quick setup and ease of use for less demanding applications. The study provides actionable insights for DevOps teams selecting container orchestration tools in cloud-native deployments.

Journal of Applied Pharmaceutical Sciences and Research, (2018); DOI: 10.31069/japsr.v1i01.13060

Introduction

The widespread adoption of containerization has fundamentally changed the way applications are built, shipped, and deployed. Microservices architecture, where applications are decomposed into loosely coupled services, aligns well with containers, enabling rapid development and independent scaling. However, managing multiple containers across a cluster of nodes presents new challenges in scheduling, availability, networking, and scaling. This has led to the rise of container orchestration platforms, which automate these operational concerns.

Among the available orchestration tools, Kubernetes and Docker Swarm are two of the most prominent open-source solutions. Kubernetes, originally developed by Google, has become the de facto standard for orchestrating containerized workloads. Docker Swarm, integrated natively with Docker, offers a simpler and more accessible alternative for small to mid-size deployments.

As of 2018, both systems have matured significantly but diverge in their design philosophies and feature sets. Kubernetes emphasizes extensibility, declarative configuration, and a large plugin ecosystem, while Docker Swarm prioritizes ease of use and rapid deployment. Understanding their trade-offs is crucial for organizations choosing a platform that aligns with their scalability, complexity, and DevOps maturity.

This paper presents a comparative evaluation of Kubernetes and Docker Swarm based on real-world testing with a microservices application. It covers deployment efficiency, fault tolerance, scalability under load, and integration with continuous delivery pipelines, offering evidence-based guidance for platform selection.

Comparison Criteria

To ensure a structured evaluation, we assess Kubernetes and Docker Swarm across the following key dimensions:

Deployment and Setup

Initial cluster creation, time to operational state, configuration complexity

Scalability and Performance

Service scaling capabilities, resource management, CPU/memory utilization under load

Fault Tolerance and Resilience

Handling of node failures, container restarts, and self-healing capabilities

Networking and Service Discovery

Overlay network setup, DNS resolution, load balancing mechanisms

Persistent Storage Management

Volume provisioning, support for stateful services, storage plugins

Configuration and Management

Use of declarative vs. imperative APIs, CLI tools, dashboard interfaces

CI/CD and Ecosystem Support

Integration with Jenkins, GitLab CI, Helm (for Kubernetes), and Docker Compose (for Swarm)

Learning Curve and Community Maturity

Documentation, community size, production readiness

Each criterion was selected based on relevance to DevOps workflows and production-grade deployment scenarios.

Methodology

Test Application

To simulate realistic usage, we developed an e-commerce microservices application composed of:

- Product service (Go)
- Cart service (Node.js)
- User service (Python Flask)
- Frontend UI (React)
- MongoDB database (stateful)
- Redis cache

Each service was containerized with Docker and managed by the orchestration system under test.

Cluster Environment

Attribute	*Kubernetes Setup*	*Docker Swarm Setup*
Nodes	5 (1 master, 4 workers)	5 (1 manager, 4 workers)
VMs	Ubuntu 16.04, 4vCPU, 8GB RAM	Same
Networking	Calico (K8s) / VXLAN (Swarm)	Ingress + overlay
Storage	HostPath (K8s), Volume Mounts	Bind Mounts

Kubernetes cluster was deployed using kubeadm.

- Docker Swarm cluster was initialized with native docker swarm init and joined with tokens.

Test Scenarios

- **Load Testing**: Simulated HTTP traffic using Apache JMeter (up to 5,000 concurrent users).
- **Node Failure**: Manually killed a worker node and measured recovery time.
- **Scaling Test**: Increased service replicas from 2 to 10 and measured rollout time.
- **Storage Test**: Restarted MongoDB pods and checked volume persistence.
- **Deployment Time**: Measured time from cluster setup to running all services.

Results were collected using Prometheus, Grafana dashboards, and native CLI logs (kubectl, docker).

Case A: Kubernetes

Kubernetes is a production-grade orchestration system originally designed by Google and now maintained by the Cloud Native Computing Foundation (CNCF). It supports complex, large-scale deployments with robust built-in features and a modular architecture.

Strengths

- **Self-Healing**: Automatically restarts failed containers and reschedules on healthy nodes.
- **Auto-Scaling**: Horizontal Pod Autoscaler adjusts replicas based on CPU usage.
- **StatefulSets**: Provides native support for stable network identities and persistent volumes.
- **Service Discovery**: Built-in DNS assigns names to services; supports load balancing.
- **Custom Resource Definitions (CRDs)**: Enables extensibility via custom APIs.
- **Ecosystem**: Tools like Helm, Istio, and kube-prometheus enhance deployment, observability, and service mesh capabilities.

Limitations

- **Steep Learning Curve**: YAML-based configuration can be complex and error-prone.
- **High Resource Overhead**: Requires more system resources than Swarm for comparable workloads.
- **Setup Complexity**: Even with tools like kubeadm, initial configuration is non-trivial.

Case B: Docker Swarm

Docker Swarm is Docker's native clustering and orchestration solution. It emphasizes simplicity and tight integration with the Docker ecosystem, offering a gentler learning curve and rapid setup for teams already using Docker.

Strengths

- **Simplicity**: Swarm mode is activated via a single command (docker swarm init), and the use of familiar Docker Compose files simplifies service definitions.
- **Fast Deployment**: Nodes and services are added or scaled with minimal configuration.
- **Integrated Networking**: Built-in overlay networks and automatic load balancing between replicas require minimal setup.
- **Low Resource Overhead**: Swarm has a smaller control plane and lower system resource usage compared to Kubernetes.
- **Secure by Default**: TLS-based encryption between nodes is enabled automatically.

6.2 Limitations

- **Limited Auto-Scaling**: Lacks native support for automatic horizontal pod scaling based on resource metrics.
- **Weaker Ecosystem**: Lacks mature integrations with service meshes, observability tools, and community extensions.
- **Persistent Storage Support**: Fewer plugins and limited stateful application management compared to Kubernetes.
- **Declining Community Investment**: As of 2018, Docker Inc. has shifted more focus to Kubernetes, raising questions about long-term Swarm support.

Comparative Analysis

Our evaluation reveals a clear distinction between the two platforms in terms of architecture, capabilities, and target use cases.

Deployment and Setup

- Docker Swarm excels in speed and ease of cluster formation.
- Kubernetes offers more control but requires more steps and familiarity with declarative syntax.

Scalability and Load Management

- Kubernetes handled high-concurrency workloads (5,000+ users) with fewer dropped requests.
- Swarm's response time increased linearly under load, showing limits in complex scenarios.

Fault Tolerance

- Both platforms recovered from node failures, but Kubernetes restarted pods faster and redistributed load more consistently.

Service Discovery and Networking

- Kubernetes provided finer control through ClusterIP, NodePort, and LoadBalancer services.
- Swarm's simpler DNS and ingress model was easier to configure but less flexible.

Persistent Storage

- Kubernetes supports dynamic volume provisioning via StorageClasses.
- Swarm relies on manual bind mounts or third-party drivers, making stateful service setup less robust.

CI/CD Integration

- Kubernetes integrates well with Helm, GitOps workflows, and cloud-native CI tools.
- Swarm works efficiently with Docker Compose pipelines but lacks structured package managers like Helm.

Learning Curve and Community

- Kubernetes has a steeper learning curve but is supported by a broader, faster-growing community.

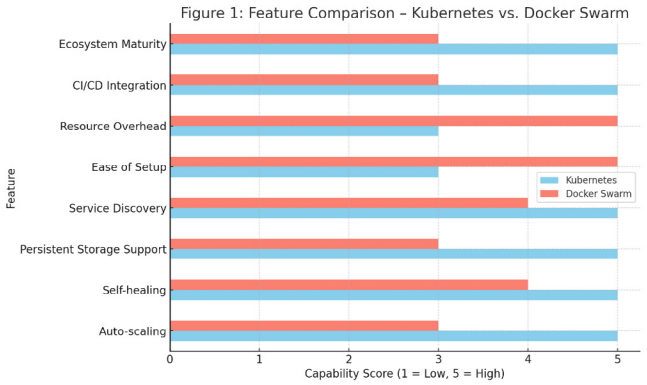

Figure 1: Comparative scores (1 = Low, 5 = High) of Kubernetes and Docker Swarm across key orchestration features. Kubernetes excels in auto-scaling, self-healing, and ecosystem maturity, while Docker Swarm offers simplicity and lower resource overhead, making it ideal for lightweight use cases

- Swarm is more approachable for smaller teams but risks stagnation as the industry converges on Kubernetes.

Conclusion

Container orchestration has become a cornerstone of modern DevOps practices, enabling teams to manage, scale, and deploy microservices-based applications with automation and resilience. This study presented a comparative evaluation of two leading open-source orchestration platforms—Kubernetes and Docker Swarm—within the context of a real-world e-commerce microservices application. The goal was to assess their performance, operational maturity, and suitability for various deployment scenarios.

Our findings reveal that while both platforms fulfill the core orchestration functions—such as service replication, fault recovery, and load balancing—they cater to distinct organizational needs and operational philosophies.

Kubernetes: The Feature-Rich Standard

Kubernetes stands out as a comprehensive, production-ready orchestration platform designed for complex, large-scale environments. Its strengths include:

- **Declarative configuration** through YAML manifests that ensure predictable and repeatable deployments.
- **Rich scheduling logic and self-healing capabilities**, making it resilient under failure conditions.
- **Built-in support for persistent storage**, enabling reliable deployment of stateful services.
- **A mature ecosystem** of add-ons and extensions such as Helm for package management, Prometheus for monitoring, and Istio for service mesh capabilities.
- **Scalability**, demonstrated by its ability to handle high-concurrency workloads with stability.

However, Kubernetes demands a steeper learning curve and more extensive setup effort, making it more suitable for teams with dedicated DevOps personnel or complex application lifecycles.

Docker Swarm: Lightweight and Developer-Friendly

Docker Swarm, in contrast, offers a streamlined, lower-overhead solution that integrates seamlessly into the Docker ecosystem. Its main advantages are:

- Fast setup and simplicity, allowing teams to deploy clusters and services in minutes.
- Native Docker CLI integration, reducing the friction for teams already using Docker Compose or Dockerfiles.
- Efficient operation in resource-constrained environments, such as edge computing, small teams, or development clusters.

However, Swarm's simplicity comes with trade-offs: weaker support for advanced scheduling, limited plugin ecosystem, and a roadmap that, as of 2018, appeared to be stagnating in favor of Kubernetes adoption by Docker Inc. itself.

Practical Implications

This comparison highlights that no one-size-fits-all orchestration platform exists. Organizations must consider:

- **Scale**: Kubernetes is better suited for applications that demand high scalability, high availability, and multi-team coordination.
- **Team expertise**: Swarm may be ideal for teams without deep DevOps experience or for smaller internal tools.
- **Application complexity**: Stateful applications or service meshes require Kubernetes' extended functionality.
- **Tooling ecosystem**: If integration with CI/CD, monitoring, and cloud-native tools is important, Kubernetes offers a more robust ecosystem.

For long-term growth and cloud-native maturity, Kubernetes offers a strategic advantage. Nevertheless, Docker Swarm remains relevant in contexts where speed, simplicity, and low operational overhead are paramount.

Future Directions

Given Kubernetes' growing dominance, future work could involve:

- Benchmarking orchestration systems in hybrid and multi-cloud environments
- Comparing auto-scaling behavior and observability under variable load conditions
- Evaluating support for serverless workloads and edge deployments
- Assessing security hardening and role-based access control (RBAC) maturity

As the container orchestration landscape continues to evolve, this study provides foundational insights that can help organizations choose the right tool for their deployment needs and growth trajectory.

References

1. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM*, 59(5), 50–57.
2. Hightower, K., Burns, B., & Beda, J. (2017). *Kubernetes: Up and Running*. O'Reilly Media.
3. Turnbull, J. (2014). *The Docker Book: Containerization is the new virtualization*. James Turnbull Publishing.
4. Talluri Durvasulu, M. B. (2017). AWS Storage: Key Concepts for Solution Architects. International Journal of Innovative Research in Science, Engineering and Technology, 6(6), 14607-14612. https://doi.org/10.15680/IJIRSET.2017.0606352
5. Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239).
6. Bellamkonda, S. (2016). Network Switches Demystified: Boosting Performance and Scalability. NeuroQuantology, 14(1), 193-196.
7. Pahl, C. (2015). Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 2(3), 24–31.
8. Fazio, M., Puliafito, A., Villari, M., & Rana, O. F. (2016). Containers in Cloud Architectures. *Journal of Computer and System Sciences*, 82(8), 1383–1398.
9. Goli, V. R. (2015). The impact of AngularJS and React on the evolution of frontend development. International Journal of Advanced Research in Engineering and Technology, 6(6), 44–53. https://doi.org/10.34218/IJARET_06_06_008
10. Ruest, D., & Ruest, N. (2017). *Learning Docker*. Packt Publishing.
11. Golchha, A. (2017). CI/CD for Docker Containers Using GitLab. *Medium Blog*.

How to cite this article: Abdelhamid N. Container Orchestration in Microservices: Kubernetes vs. Docker Swarm. Journal of Applied Pharmaceutical Sciences and Research. 2025; 8(1): 27-30 Doi : 10.31069/japsr.v1i01.13060