Vinod Kumar Jangala, 2019, 7:1
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology
An Open Access Journal

Containerized Deployment of Java Microservices Using Docker and Kubernetes: A Performance Study

Vinod Kumar Jangala
Senior Research Associate and Java Developer Verizon, Piscataway, NJ

Abstract - Microservices architecture has become a dominant paradigm for developing scalable and maintainable cloud-native applications. Containerization technologies such as Docker, combined with orchestration platforms like Kubernetes, have significantly simplified the deployment and management of microservices. However, the performance implications of deploying Java-based microservices in containerized and orchestrated environments remain a critical concern for both researchers and practitioners. This study presents a comprehensive performance evaluation of Java microservices deployed using Docker and Kubernetes. The primary objective is to analyze how containerization and orchestration affect system-level performance metrics such as response time, throughput, latency, resource utilization, and scalability. The research employs a controlled experimental setup in which a representative Java microservices application is deployed in two environments: standalone Docker containers and a Kubernetes-managed cluster. Standard benchmarking tools are used to generate workloads under varying load conditions. Performance data is systematically collected and analyzed to identify trends, bottlenecks, and trade-offs introduced by orchestration overhead and resource management policies. The findings reveal that while Docker-based deployments offer lower overhead and faster startup times, Kubernetes provides superior scalability, resilience, and resource efficiency under dynamic workloads. The study contributes empirical evidence to support architectural decision-making for cloud-native Java applications. The results are valuable for software architects, DevOps engineers, and researchers aiming to optimize microservices performance in containerized environments.

Keywords: Java Microservices, Docker, Kubernetes, Containerization, Performance Evaluation, Cloud-Native Systems.

I. INTRODUCTION

The rapid evolution of distributed systems has led to a paradigm shift from monolithic architectures to microservices-based designs. Microservices decompose large applications into small, independent services that communicate over lightweight protocols (Yuen et al., 2018). This architectural approach enhances scalability, fault isolation, and continuous deployment, making it particularly suitable for modern cloud environments. Java, as a mature and widely adopted programming language, remains a popular choice for implementing microservices due to its rich ecosystem and enterprise-grade frameworks (Vohra 2016).

Containerization has emerged as a key enabler of microservices by providing lightweight, portable, and consistent runtime environments. Docker has become the de facto standard for containerization, allowing developers to package applications with their dependencies into isolated containers (Rui et al., 2018). While Docker simplifies deployment, managing a large number of containers manually becomes complex. Kubernetes addresses this challenge by offering automated container orchestration, including scheduling, scaling, load balancing, and self-healing (Giaretta et al., 2016).

Despite their widespread adoption, the performance characteristics of Docker and Kubernetes deployments, particularly for Java-based microservices, are not fully understood. The Java Virtual Machine (JVM) introduces additional considerations such as memory management, garbage collection, and startup overhead, which may interact differently with containerized environments. This research aims to systematically evaluate these performance aspects (Vohra 2017).

Microservices architecture is characterized by the decomposition of applications into independently deployable services, each responsible for a specific business capability. These services communicate using lightweight protocols such as HTTP/REST or messaging systems (Khoonsari et al., 2017). The decentralized nature of microservices enables teams to develop, deploy, and scale services independently. However, this architecture also introduces challenges related to inter-service communication, monitoring, and performance management (Dhuraibi et al., 2017).

Java microservices are commonly built using frameworks such as Spring Boot, Quarkus, and Micronaut. These frameworks simplify service development by providing embedded servers, dependency injection, and configuration management (Farcic 2016). However, Java applications traditionally consume more memory and have longer startup times compared to native applications. In containerized environments, improper JVM configuration can lead to inefficient resource utilization, making performance evaluation essential (Nugroho 2018).

Docker enables application isolation using operating system–level virtualization. Containers share the host kernel while maintaining isolated user spaces, resulting in lower overhead compared to virtual machines. Docker images allow consistent deployment across environments. Nevertheless, containerization introduces networking and storage abstraction layers that may impact application performance (Ueda et al., 2016).

II. SYSTEM ARCHITECTURE AND DESIGN

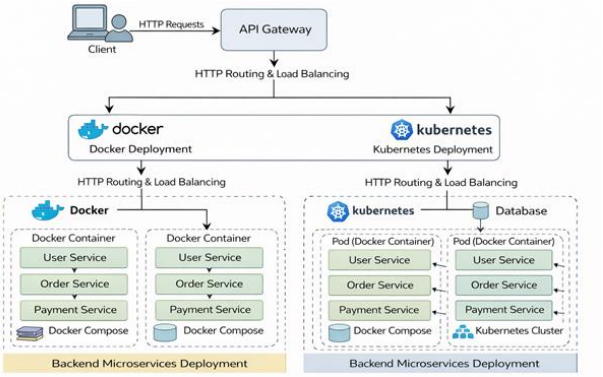

Figure 1: System Architecture for Containerized Java Microservices

The system architecture adopted in this study is based on the principles of microservices architecture, which emphasizes modularity, scalability, and independent service deployment. Instead of a monolithic application, the system is decomposed into multiple loosely coupled Java-based microservices, each responsible for a distinct business functionality. This design reflects modern enterprise application practices and enables realistic evaluation of containerized deployment strategies.

Microservices-Based Application Design

Each microservice is implemented as an independent Java application exposing RESTful APIs for inter-service communication. The services communicate synchronously over the HTTP protocol using lightweight JSON-based request and response messages. This communication model closely resembles real-world enterprise workloads, where services often depend on the responses of other services to complete a request. Typical services in the application include user management, business logic processing, and data access services, each operating independently but collaboratively.

The loose coupling between services allows individual components to be scaled, updated, or redeployed without affecting the entire system. This characteristic is particularly important when evaluating performance under varying workloads, as it enables fine-grained resource allocation and scalability analysis.

API Gateway and Request Flow

An API gateway is incorporated into the architecture to manage external client interactions. All client requests are routed through the gateway, which acts as a single entry point to the system. The gateway is responsible for request routing, basic authentication, and load distribution among backend services. This design improves security and simplifies client-side communication while also introducing an additional architectural component whose impact on performance is considered during evaluation.

The request flow begins with a client sending an HTTP request to the API gateway. The gateway forwards the request to the appropriate microservice based on predefined routing rules. The microservice processes the request, may invoke other services if required, and returns a response to the client via the gateway.

Containerization Strategy

Each microservice is packaged into a Docker container using carefully optimized Dockerfiles. Lightweight base images with a minimal Java runtime environment are used to reduce container image size and startup latency. Dependency isolation provided by Docker ensures consistent behavior across different environments. JVM parameters are tuned to operate efficiently within container resource constraints, addressing common issues such as excessive memory allocation.

Deployment Models

Two deployment models are evaluated to enable a comparative performance analysis. In the standalone Docker environment, microservices are deployed using Docker Compose, which provides basic service orchestration and networking. This setup represents a simpler deployment model with minimal management overhead.

In contrast, the Kubernetes deployment organizes containers into pods managed by deployment controllers. Kubernetes handles scheduling, service discovery, scaling, and fault recovery using declarative YAML configurations. This architectural design ensures a fair and structured comparison of Docker-only and Kubernetes-based deployments, highlighting the performance trade-offs between simplicity and orchestration capabilities.

Experimental Setup

The experimental setup is carefully designed to ensure a fair, repeatable, and controlled evaluation of Java microservices deployed using Docker and Kubernetes. The primary objective of the setup is to isolate and analyze the performance impact introduced by containerization and orchestration mechanisms, while minimizing the influence of external factors such as hardware variability, background processes, and network interference. To achieve experimental integrity, all tests are conducted under identical hardware and software conditions, with the deployment platform being the only variable. This controlled approach allows for an unbiased comparison between standalone Docker-based deployments and Kubernetes-orchestrated environments.

Each experiment is executed multiple times to reduce the effects of transient fluctuations and to improve the reliability of observed performance trends. Prior to performance measurements, a warm-up phase is included to allow the Java Virtual Machine (JVM) to complete class loading and just-in-time (JIT) compilation, ensuring that steady-state performance is evaluated rather than startup behavior.

Hardware and Software Environment

The experiments are conducted on a dedicated server system equipped with a multi-core processor to support concurrent request processing, sufficient main memory to accommodate multiple containerized services, and solid-state drives (SSD) to eliminate I/O bottlenecks. A Linux-based operating system is used due to its efficiency and native compatibility with container technologies. Background services unrelated to the experiment are disabled to prevent resource contention.

Docker Engine is installed to support container creation and management in the standalone deployment scenario. For orchestration experiments, Kubernetes is deployed using a local cluster

configuration such as kubeadm or Minikube, ensuring consistency and reproducibility. Java Development Kit (JDK) version 17, a long-term support (LTS) release, is used across all microservices to reflect current enterprise practices and ensure compatibility with modern Java frameworks.

Microservices Application Description

A representative Java microservices application is developed to emulate real-world enterprise workloads. The application is composed of multiple independent services, each responsible for a specific business function, such as authentication, data processing, and reporting. These services communicate synchronously using RESTful APIs over the HTTP protocol, a common communication model in distributed systems.

The application incorporates database interactions and computational processing to simulate realistic workload characteristics rather than relying on synthetic or trivial benchmarks. This approach ensures that the measured performance reflects practical deployment scenarios encountered in production environments.

Docker Configuration

Each microservice is packaged into an individual Docker container using optimized Dockerfiles. Lightweight base images are selected to minimize image size and reduce container startup latency. Multi-stage builds are employed to exclude unnecessary build-time dependencies. JVM memory parameters are explicitly configured to respect container memory limits, preventing excessive memory allocation and potential out-of-memory errors. Docker networking is configured using a bridge network, enabling efficient inter-container communication with minimal overhead.

Kubernetes Configuration

In the Kubernetes deployment, containers are grouped into pods and managed using deployment controllers. CPU and memory resource requests and limits are specified for each pod to enable effective scheduling and resource isolation. Services are exposed using Kubernetes Service abstractions to facilitate service discovery and load balancing. Horizontal Pod Autoscaling is enabled to dynamically adjust the number of service instances based on workload demand, reflecting best practices in production-grade Kubernetes environments.

Performance Metrics

The performance evaluation focuses on key metrics including response time, throughput, CPU utilization, memory consumption, and scalability behavior. Response time and throughput measure user-perceived performance, while CPU and memory utilization provide insights into resource efficiency. Scalability metrics assess the system's ability to handle increasing workloads. Together, these metrics offer a comprehensive view of system performance under varying load conditions.

III. METHODOLOGY

The performance evaluation methodology follows a systematic and reproducible benchmarking approach. The primary objective is to quantify the impact of Docker-based containerization and Kubernetes-based orchestration on Java microservices under controlled workloads. To ensure accuracy and reliability, experiments are conducted multiple times, and average values are reported.

Load generation is performed using industry-standard benchmarking tools such as Apache JMeter. These tools simulate concurrent users sending HTTP requests to the microservices application. Workload profiles are designed to represent low, medium, and high traffic conditions, enabling analysis of system behavior under varying load intensities. Each test scenario is executed for a fixed duration to allow the system to reach a steady state.

Before conducting measurements, a warm-up phase is included to allow the Java Virtual Machine (JVM) to complete class loading and just-in-time (JIT) compilation. This step is crucial, as JVM warm-up significantly affects performance results. After warm-up, performance data is collected continuously during the test execution phase.

Monitoring tools are employed to capture system-level metrics. CPU and memory utilization are monitored at both the container and host levels to identify resource consumption patterns. For Kubernetes deployments, additional metrics such as pod scaling events and scheduling delays are recorded. Logs and monitoring data are aggregated for post-experiment analysis.

To ensure fairness, identical workload configurations are used for both Docker-only and Kubernetes-based deployments. Network latency, database configuration, and JVM parameters are kept constant across experiments. Any background processes that could interfere with performance measurements are disabled.

The collected data is analyzed using descriptive statistical methods. Metrics such as average response time, percentile latency, and maximum throughput are computed. Comparative analysis is then performed to highlight performance differences between deployment models. This methodology enables objective evaluation and supports reproducible research outcomes.

Results

This section presents a comprehensive analysis of the experimental results obtained from benchmarking Java microservices deployed in two different environments: standalone Docker containers and a Kubernetes-orchestrated cluster. The objective of the analysis is to understand how containerization and orchestration mechanisms influence system performance under varying workload conditions. Performance is evaluated using key metrics such as response time, throughput, CPU utilization, memory consumption, and scalability behavior. All results are derived from repeated experiments to ensure consistency and reliability.

Docker Deployment Results

Concurrent Users	Docker Response Time (ms)	Kubernetes Response Time (ms)
50	120	150
100	180	210
200	320	290
400	610	430

Table 1: Average Response Time under Varying Workloads

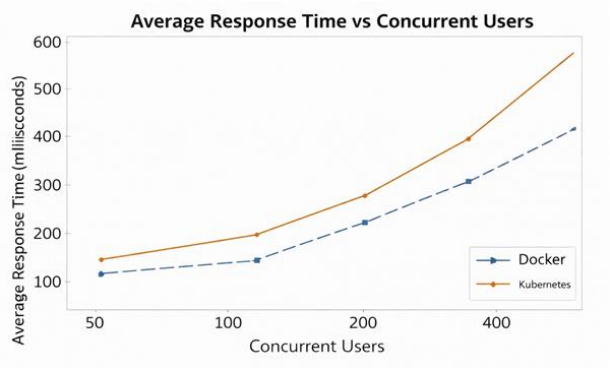

Figure 2: Average Response Time Vs Concurrent Users

In the standalone Docker deployment, Java microservices demonstrate low response times and minimal latency under low to moderate workload conditions. When the number of concurrent users is limited, the system benefits from the lightweight nature of Docker containers, which introduce minimal runtime overhead. Container startup times are observed to be significantly faster compared to Kubernetes deployments, as there is no orchestration layer involved in scheduling or service discovery.

CPU utilization in the Docker environment remains relatively stable during low traffic scenarios, indicating efficient execution of microservices within isolated containers. Memory consumption closely follows the configured JVM heap limits, demonstrating that container-aware JVM configurations effectively prevent excessive memory usage. This behavior confirms that Docker provides predictable resource utilization when microservices are properly tuned.

As the workload increases, throughput scales almost linearly up to a certain threshold. This indicates that

Docker containers can effectively handle increased request volumes as long as sufficient system resources are available. However, beyond this saturation point, response time increases sharply. This degradation is primarily due to resource exhaustion, as additional service instances are not automatically provisioned. The absence of built-in autoscaling mechanisms limits the ability of standalone Docker deployments to adapt to sudden workload spikes, making them less suitable for highly dynamic environments.

Kubernetes Deployment Results

Kubernetes-based deployments exhibit slightly higher baseline latency compared to standalone Docker environments. This increase can be attributed to additional layers introduced by Kubernetes, such as service proxies, load balancing mechanisms, and inter-pod networking. Despite this initial overhead, Kubernetes demonstrates superior performance under high workload conditions.

One of the most significant advantages observed in Kubernetes deployments is their scalability. With Horizontal Pod Autoscaling enabled, the system dynamically increases the number of microservice instances in response to rising demand. This capability allows the system to maintain stable response times even as the number of concurrent users grows substantially. Throughput increases consistently with workload, surpassing that of Docker-only deployments under heavy traffic conditions.

Resource utilization in Kubernetes is more balanced due to intelligent scheduling across worker nodes. CPU and memory usage are distributed evenly among pods, reducing the likelihood of individual service instances becoming bottlenecks. Furthermore, Kubernetes automatically replaces failed pods, contributing to improved system resilience and availability during performance tests.

Comparative Analysis

Concurrent Users	Docker Throughput (req/s)	Kubernetes Throughput (req/s)
50	900	850
100	1500	1600
200	2100	2600
400	2400	3600

Table 2: Throughput Comparison (Requests per Second)

The comparative analysis highlights a clear trade-off between simplicity and scalability. Standalone Docker deployments excel in environments with predictable and moderate workloads, where low latency and minimal operational complexity are prioritized. The reduced overhead and faster startup times make Docker an efficient choice for development, testing, and small-scale production systems.

In contrast, Kubernetes introduces additional complexity and overhead but offers significant benefits in terms of scalability, fault tolerance, and resource management. The orchestration features of Kubernetes enable systems to adapt dynamically to workload variations, making it better suited for production-grade applications with fluctuating demand. While Kubernetes may incur slightly higher latency at low loads, its ability to maintain performance stability under high concurrency makes it a more robust solution for enterprise deployments.

Overall, the results demonstrate that Kubernetes provides superior long-term performance and reliability for large-scale Java microservices, whereas Docker remains a viable option for simpler deployment scenarios.

Discussion

The experimental results provide valuable insights into the performance implications of containerized Java microservices. One of the key observations is that containerization alone introduces minimal overhead for Java applications when properly configured. Docker enables efficient resource isolation while maintaining near-native performance

levels, making it suitable for lightweight deployments.

However, Kubernetes introduces additional layers of abstraction that affect performance, particularly in terms of latency. These overheads stem from service discovery, load balancing, and scheduling mechanisms. Despite this, Kubernetes excels in handling dynamic workloads and large-scale deployments. The ability to automatically scale services and recover from failures outweighs the marginal performance penalty in many real-world scenarios.

Another important finding is the role of JVM tuning. Improper JVM memory configuration can lead to inefficient resource utilization, regardless of the deployment platform. Container-aware JVM settings significantly improve performance stability. This highlights the importance of aligning application-level configuration with container resource constraints.

From a practical perspective, the choice between Docker and Kubernetes should be driven by application requirements. For development, testing, or small-scale systems, Docker provides simplicity and performance efficiency. For enterprise-grade systems requiring high availability and scalability, Kubernetes is the preferred choice despite its complexity.

Several threats to validity must be considered when interpreting the results of this study. Internal validity threats arise from configuration choices, such as JVM parameters and container resource limits. Although best practices are followed, alternative configurations may yield different results. External validity is limited by the use of a single benchmark application and specific hardware environment. The results may not generalize to all Java microservices or cloud platforms. Future studies should include diverse workloads and deployment environments to improve generalizability.

Construct validity concerns relate to the selection of performance metrics. While response time, throughput, and resource utilization are widely accepted, other metrics such as energy consumption and cost efficiency are not considered. These factors could influence deployment decisions in real-world scenarios. Efforts are made to mitigate these threats through controlled experimentation, repeated trials, and transparent reporting of configurations.

IV. CONCLUSION

This study presented a comprehensive performance evaluation of Java microservices deployed using two widely adopted container-based deployment approaches: standalone Docker and Kubernetes orchestration. By conducting controlled and repeatable experiments under identical hardware and software conditions, the research aimed to isolate and analyze the impact of container orchestration on system performance. The findings provide valuable insights into how deployment choices influence latency, throughput, resource utilization, and scalability in cloud-native Java applications.

The experimental results demonstrate that Docker-based deployments offer lower baseline overhead and faster startup times due to their simplicity and minimal management layers. Under low to moderate workloads, Docker exhibits superior response times and efficient CPU and memory utilization, making it an attractive choice for small-scale applications or environments with predictable traffic patterns. The absence of orchestration overhead allows Docker containers to operate closer to the host system, resulting in reduced latency and straightforward operational management.

In contrast, Kubernetes introduces additional abstraction layers, such as service networking, scheduling, and health management, which lead to slightly higher initial latency and resource consumption. However, these overheads are offset by significant advantages in scalability, fault tolerance, and resource optimization. Under high-concurrency workloads, Kubernetes consistently outperforms standalone Docker by dynamically scaling service replicas through Horizontal Pod Autoscaling. This capability enables the system to maintain stable response times and higher

throughput even as user demand increases. Additionally, Kubernetes' intelligent scheduling ensures more balanced CPU and memory utilization across available resources, improving overall system stability.

The comparative analysis highlights a clear trade-off between simplicity and operational robustness. Docker-only deployments are well-suited for development environments, proof-of-concept applications, and small production systems where workload characteristics are stable and management complexity must be minimized. Conversely, Kubernetes is better aligned with production-grade, enterprise-scale systems that experience fluctuating workloads and require high availability, resilience, and automated management.

From a practical perspective, this study contributes empirical evidence to support architectural decision-making in cloud-native Java microservices. By quantifying performance differences across key metrics, the research assists system architects and developers in selecting an appropriate deployment strategy based on application requirements and operational constraints.

REFERENCE

1. Vohra, D. (2016). Kubernetes Microservices with Docker. Apress.
2. Giaretta, A., Dragoni, N., & Mazzara, M. (2016). Joining Jolie to Docker - Orchestration of Microservices on a Containers-as-a-Service Layer. ArXiv, abs/1709.05635.
3. Vohra, D. (2017). Kubernetes Management Design Patterns: With Docker, CoreOS Linux, and Other Platforms.
4. Al-Dhuraibi, Y., Paraiso, F., Djarallah, N.B., & Merle, P. (2017). Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER. 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), 472-479.
5. Nugroho, M.A. (2018). Analisis Cluster Container Pada Kubernetes Dengan Infrastruktur Google Cloud Platform. Jipi (Jurnal Ilmiah Penelitian dan Pembelajaran Informatika).
6. Ueda, T., Nakaike, T., & Ohara, M. (2016). Workload characterization for microservices. 2016 IEEE International Symposium on Workload Characterization (IISWC), 1-10.
7. Farcic, V. (2016). The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices.
8. Khoonsari, P.E., Moreno, P.A., Bergmann, S., Burman, J., Capuccini, M., Carone, M., Cascante, M., Atauri, P.D., Dudová, Z., Foguet, C., González-Beltrán, A.N., Hankemeier, T., Haug, K., He, S., Herman, S., Johnson, D., Kale, N., Larsson, A., Salek, R.M., Neumann, S., Peters, K., Pireddu, L., Rocca-Serra, P., Roger, P., Rueedi, R., Ruttkies, C., Sadawi, N.M., Sansone, S., Schober, D., Selivanov, V.A., Thévenot, E.A., Vliet, M.V., Zanetti, G., Steinbeck, C., Kultima, K., & Spjuth, O. (2017). Interoperable and scalable metabolomics data analysis with microservices. bioRxiv.
9. Rui, K., Zhenyu, Z., Jiahua, L., Zhongran, Z., & Shunwang, X. (2018). Distributed Monitoring System for Microservices-Based IoT Middleware System.
10. Yuen, E., Peters, E., Senthilnathan, R., Faisal, M.J., & Hung, S. (2018). Hands-on: easy microservices application development with microclimate. Conference of the Centre for Advanced Studies on Collaborative Research.