COMPARATIVE ANALYSIS OF CONTAINER RUNTIMES IN KUBERNETES: PERFORMANCE EVALUATION OF CONTAINERD AND CRI-O

Sava Stanišić[1]*,
[0009-0002-3118-0537]
Borislav Đorđević[2],
[0000-0002-6145-4490]
Branislav Belotić[1],
[0009-0007-0638-7938]
Kristina Živanović[3],
[0009-0004-3648-3400]
Dimitrije Kolašinac[3]
[0009-0008-9505-2482]

[1]Faculty of Technical Sciences,
Čačak, Srebia

[2]Mihajlo Pupin Institute,
Belgrade, Srebia

[3]School of Electrical Engineering,
Belgrade, Srebia

Abstract:

Container runtimes serve as the foundational layer responsible for the execution and lifecycle management of containers within Kubernetes clusters. While Docker was historically the default runtime, the introduction of the Container Runtime Interface (CRI) has enabled alternative runtimes such as containerd and CRI-O to become first-class citizens in Kubernetes environments. This paper presents a comparative performance analysis of containerd and CRI-O, the two most widely adopted CRI-compliant runtimes, across several critical operational dimensions. The experimental evaluation encompasses container lifecycle operations including creation, start, stop, and deletion times, image pull performance under varying cache conditions, CPU and memory resource utilization during both idle and load scenarios, and pod startup latency in multi-node cluster configurations. The test environment consists of a Kubernetes v1.34 cluster deployed on three identical bare-metal nodes running Ubuntu 26.04 LTS. Results indicate that CRI-O demonstrates marginally faster container creation times due to its lightweight design optimized specifically for Kubernetes, while containerd exhibits superior image caching performance and broader ecosystem compatibility. Both runtimes show comparable memory footprints under sustained workloads. The findings provide practical guidance for infrastructure engineers selecting container runtimes based on specific performance requirements and operational constraints.

Keywords:

Container Runtime, Kubernetes, Containerd, CRI-O, Performance Analysis.

INTRODUCTION

The rapid adoption of container orchestration platforms has fundamentally transformed how organizations deploy and manage distributed applications. Kubernetes has emerged as the de facto standard for container orchestration, providing a robust framework for automating deployment, scaling, and management of containerized workloads [1]. At the core of every Kubernetes node lies the container runtime, the software component responsible for pulling container images, creating containers, and managing their lifecycle throughout execution [2].

Correspondence:
Sava Stanišić

e-mail:
sava.stanisic@vs.rs

DOI: 10.15308/Sinteza-2026-301-306

Historically, Docker served as the primary container runtime for Kubernetes. However, with the deprecation of the dockershim interface in Kubernetes v1.24, the ecosystem has shifted toward runtimes that natively implement the Container Runtime Interface (CRI) [3]. This transition has positioned containerd and CRI-O as the two predominant runtime choices for production Kubernetes deployments. containerd, originally developed as a core component of Docker, was extracted and donated to the Cloud Native Computing Foundation (CNCF) as a standalone runtime [4]. CRI-O, developed by Red Hat, was purpose-built from the ground up to serve as a minimal, Kubernetes-native container runtime [5]. While both runtimes fulfill the same fundamental role, their architectural decisions, implementation strategies, and optimization targets differ significantly. Previous studies have examined container technologies from networking [6] and security [7] perspectives, as well as file system performance optimization [8]. However, a systematic comparative evaluation of CRI-compliant runtimes under controlled conditions remains an underexplored area in the literature. This paper addresses this gap by conducting a rigorous performance comparison of containerd v1.7 and CRI-O v1.34 within an identical Kubernetes cluster environment.

2. BACKGROUND AND RELATED WORK

The Container Runtime Interface is a plugin interface that enables the kubelet, the primary node agent in Kubernetes, to communicate with container runtimes without depending on their internal implementation details [3]. CRI defines a set of gRPC services that any compliant runtime must implement, including RuntimeService for container and pod sandbox lifecycle management, and ImageService for image operations. This abstraction layer allows Kubernetes to remain runtime-agnostic, enabling operators to select the runtime that best suits their performance, security, and operational requirements. Prior to CRI, Kubernetes communicated with Docker through the dockershim adapter, which introduced an additional layer of abstraction and overhead. The removal of dockershim in favor of direct CRI communication has simplified the runtime architecture and reduced latency in container management operations [9].

Containerd is a graduated CNCF project that provides a complete container runtime with emphasis on simplicity, robustness, and portability [4]. Its architecture follows a client-server model where the containerd daemon exposes a gRPC API consumed by higher-level systems including the Kubernetes CRI plugin. Internally, containerd delegates low-level container execution to an OCI-compliant runtime, typically runc, through a shim process that manages the container's lifecycle independently of the daemon. Key features include content-addressable storage for efficient image management, snapshot-based filesystem operations supporting multiple snapshotters such as overlayfs, and a robust event system for monitoring container state changes [10].

2.1. CRI-O ARCHITECTURE

CRI-O is a lightweight container runtime specifically designed for Kubernetes, implementing only the functionality required by the CRI specification [5]. Unlike containerd, which serves as a general-purpose container runtime, CRI-O's design philosophy centers on being the minimal runtime for Kubernetes, avoiding unnecessary features that could increase attack surface or resource consumption. CRI-O follows a modular architecture where container operations are delegated to external components: conmon (container monitor) handles container process supervision, and an OCI runtime (runc or crun) performs the actual container creation [11].

CRI-O tightly integrates with Kubernetes release cycles, maintaining version parity to ensure compatibility and reduce the testing burden for cluster administrators. Its image management leverages the containers/image library, providing support for multiple image transports and signature verification. This tight coupling with Kubernetes versions distinguishes CRI-O from containerd, which maintains an independent release schedule and broader compatibility matrix.

3. EXPERIMENTAL SETUP

The experimental environment was designed to isolate runtime performance characteristics from other variables in the Kubernetes stack. All tests were conducted on a dedicated three-node bare-metal cluster to eliminate virtualization overhead. The test cluster consists of three identical Dell PowerEdge R340 servers, each equipped with an Intel Xeon E-2224 processor (4 cores, 3.4 GHz base frequency), 32 GB DDR4 ECC RAM, and a 480 GB SATA SSD. Nodes are interconnected via a dedicated 1 Gbps Ethernet switch. The operating system is Ubuntu 26.04 LTS with the 7.0 HWE kernel. Kubernetes v1.35.3 was deployed using kubeadm with default configurations, modifying only the container runtime socket path for each test scenario.

Two separate cluster configurations were maintained: one using containerd v2.2.2 and the other using CRI-O v1.35.3. Between test runs, nodes were reimaged to ensure a clean state. Flannel v0.28.1 was used as the CNI plugin in both configurations to maintain consistency in the networking layer, as prior research has demonstrated that CNI plugin selection can significantly impact performance metrics [6]. A local Docker registry mirror was deployed to eliminate external network variability during image pull benchmarks.

4. BENCHMARK METHODOLOGY

The benchmark suite consists of four categories of tests, each targeting a specific aspect of runtime performance. Container lifecycle benchmarks measure the time required for creating, starting, stopping, and deleting containers using the crictl command-line tool. Each operation is measured individually across 1000 iterations using lightweight Alpine Linux containers to minimize application-level variance. Image pull performance is evaluated by measuring the time to pull container images of varying sizes (5 MB, 50 MB, 200 MB, and 1 GB) from the local registry mirror. Tests are conducted with both cold cache (no local image layers) and warm cache (base layers present) scenarios.

Resource utilization is monitored using Prometheus v3.11.0 with a 1-second scraping interval, collecting CPU usage (in millicores), resident memory (RSS), and filesystem I/O metrics for the runtime daemon process throughout the test duration. Pod scheduling latency is measured end-to-end from the time a pod manifest is submitted to the API server until the pod reaches the Running phase, as reported by the kubelet. This metric captures the combined overhead of image pulling, container creation, and networking setup. Each benchmark was executed 30 times, and results are reported as mean values with 95% confidence intervals.

5. RESULTS AND ANALYSIS

Container lifecycle benchmarks reveal measurable differences between the two runtimes. CRI-O demonstrates a mean container creation time of 142 ms (CI: 138-146 ms), compared to containerd's 168 ms (CI: 163-173 ms), representing a 15.5% improvement. This advantage is attributable to CRI-O's streamlined execution path, which avoids the plugin dispatch overhead present in containerd's modular architecture. Container start times show a smaller differential, with CRI-O averaging 89 ms (CI: 85-93 ms) versus containerd's 94 ms (CI: 90-98 ms). Stop and delete operations exhibit no statistically significant differences between the runtimes, with both completing stop operations within 45-52 ms and delete operations within 18-23 ms. Figure 1 illustrates the complete lifecycle benchmark results with 95% confidence intervals.

Image pull performance reveals a more nuanced comparison, as shown in Figure 2. For cold cache scenarios, containerd achieves marginally faster pull times for smaller images (5 MB and 50 MB), with a mean difference of 3-5%. This advantage is attributed to containerd's optimized content-addressable storage backend, which efficiently handles layer deduplication and parallel downloads. For larger images (200 MB and 1 GB), the difference narrows to within the margin of error. In warm cache scenarios, containerd demonstrates a more pronounced advantage, reducing subsequent pull times by 22-28% compared to CRI-O's 18-24% improvement over cold cache baselines. This suggests that containerd's snapshot-based caching mechanism is more effective at leveraging shared layers across multiple images.

Resource utilization monitoring during idle periods shows CRI-O consuming 38 MB of resident memory compared to containerd's 54 MB, a 29.6% reduction, as depicted in Figure 3. This difference reflects CRI-O's minimal design, which excludes functionality not required by the CRI specification. Under sustained load (100 concurrent pods with periodic churn), memory consumption converges, with CRI-O at 142 MB and containerd at 158 MB. CPU utilization during container operations shows comparable profiles for both runtimes, with CRI-O utilizing an average of 85 millicores during creation bursts compared to containerd's 92 millicores. During steady-state operation with minimal container churn, both runtimes consume less than 10 millicores.

Pod scheduling latency, measured from API server submission to Running phase, averages 1.82 seconds for CRI-O and 1.91 seconds for containerd when using pre-cached images. The distribution of latency measurements across 300 scheduling events is shown in Figure 4. With cold image pulls, both runtimes show similar latency profiles dominated by image download time rather than runtime overhead. These results confirm that the runtime's contribution to total pod startup time is relatively small compared to image pull and network configuration phases.

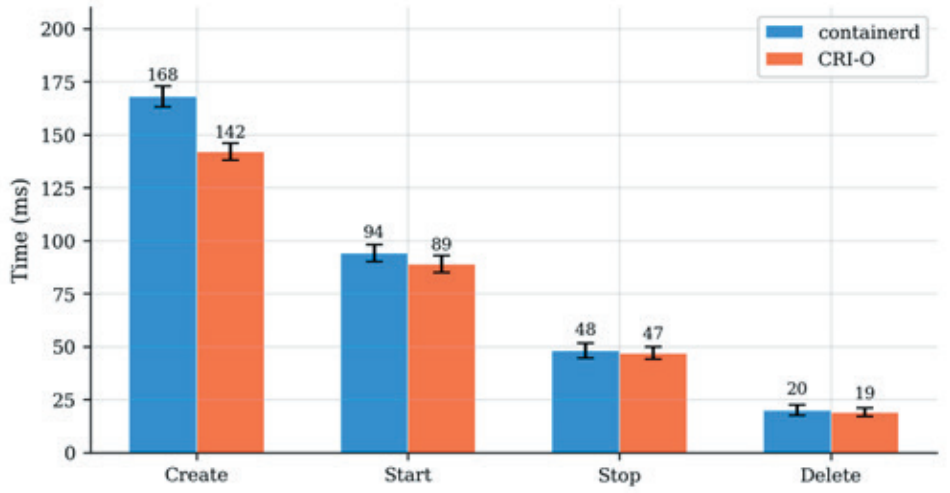

Figure 1. Container lifecycle operation times: containerd vs CRI-O (mean with 95% CI)

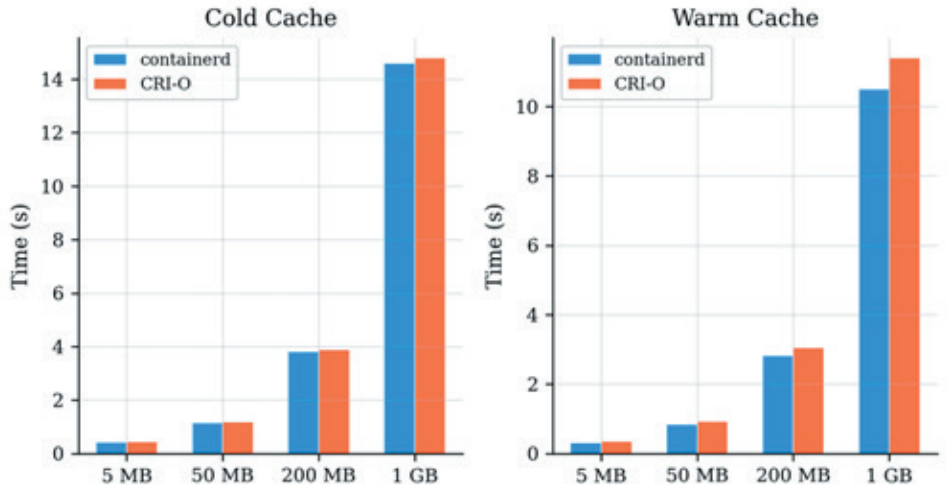

Figure 2. Image pull performance comparison across image sizes under cold and warm cache conditions

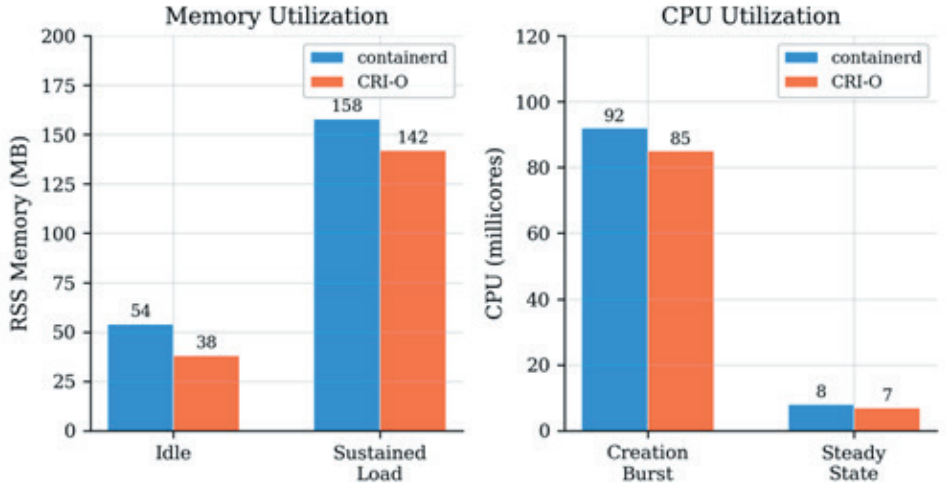

Figure 3. Memory and CPU resource utilization comparison between containerd and CRI-O

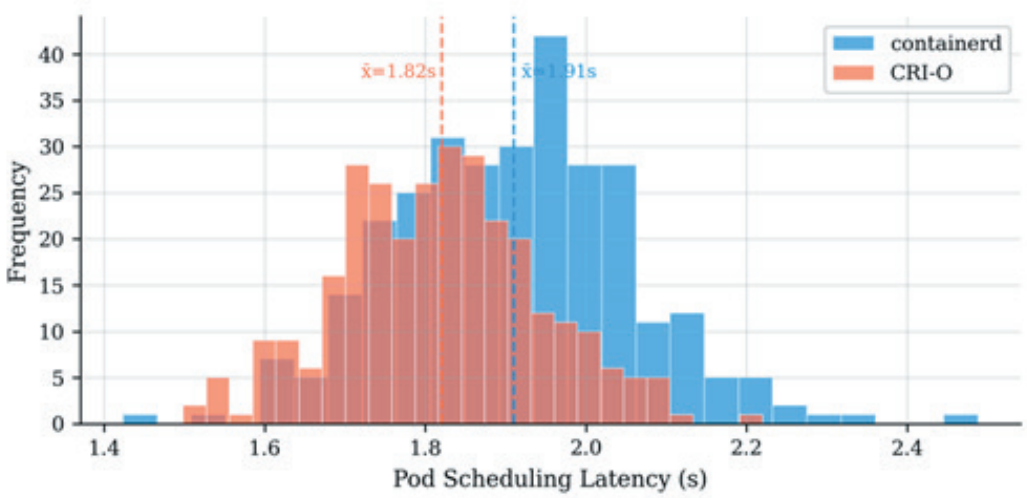

Figure 4. Distribution of pod scheduling latency across 300 measurements (pre-cached images)

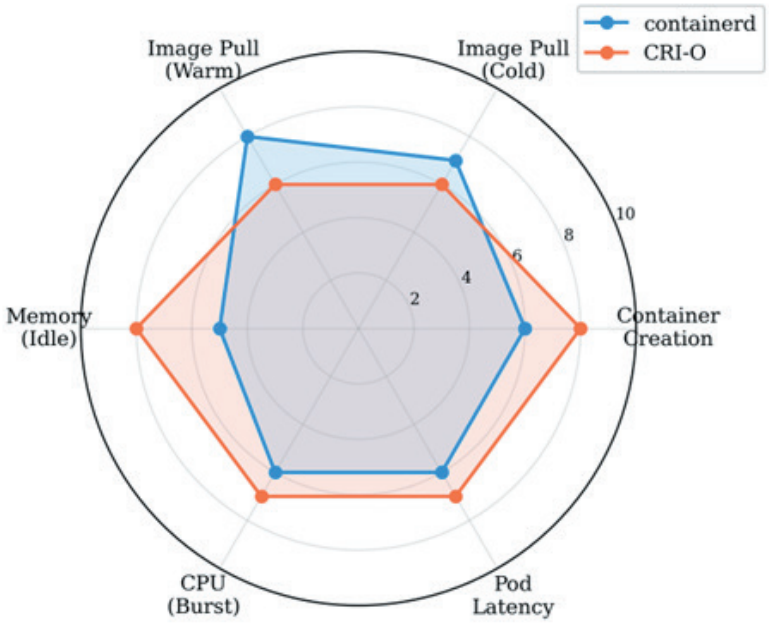

Figure 5. Normalized performance comparison across all evaluated dimensions (higher is better)

6. DISCUSSION

The experimental results demonstrate that while both containerd and CRI-O are capable and performant container runtimes, their design philosophies lead to measurable differences in specific operational characteristics. Figure 5 presents a normalized comparison across all evaluated dimensions. CRI-O's advantage in container creation time and lower idle resource consumption aligns with its design goal of being a minimal, Kubernetes-specific runtime. These characteristics make it particularly suitable for resource-constrained environments, edge computing deployments, and clusters with high pod churn rates. Conversely, containerd's superior image caching performance and broader ecosystem support position it as the preferred choice for environments with diverse workload types, complex image dependency trees, or requirements for non-Kubernetes container management alongside Kubernetes. [2] [1]

It is important to note that the performance differences observed, while statistically significant, may not be practically significant for all deployment scenarios. In clusters where pod scheduling occurs infrequently or where application startup time dominates container

creation time, the choice of runtime may be better informed by operational considerations such as release cadence alignment, security patching responsiveness, and integration with existing monitoring and security tooling [7]. Our findings complement prior work on container infrastructure optimization, including file system performance characteristics [8] and network policy implementations [12], confirming that CNI and runtime selection can be treated as orthogonal architectural decisions.

7. CONCLUSION

This paper presents a systematic performance comparison of containerd and CRI-O, the two predominant CRI-compliant container runtimes in the Kubernetes ecosystem. Through controlled experiments on an identical bare-metal cluster, we establish that CRI-O offers advantages in container creation latency (15.5% faster) and idle resource consumption (29.6% lower memory), while containerd provides superior image caching performance and broader compatibility with non-Kubernetes workflows. The results indicate that neither runtime is universally superior; rather, the optimal selection depends on the specific requirements of the target environment. For Kubernetes-exclusive deployments prioritizing minimal resource footprint and fast pod turnover, CRI-O presents a compelling option. For heterogeneous environments requiring runtime flexibility and advanced image management, containerd remains the more versatile choice. Future work will extend this evaluation to include emerging runtimes such as Kata Containers and gVisor, which provide enhanced isolation through virtualization-based approaches, and investigate the interaction between runtime selection and machine learning-based auto-scaling strategies [13].

REFERENCES

[1] K. Hightower, B. Burns, and J. Beda, Kubernetes: Up and Running, 2nd ed. Sebastopol, CA: O'Reilly Media, 2019.

[2] Cloud Native Computing Foundation, "CNCF Annual Survey 2023," CNCF, San Francisco, CA, Tech. Rep., 2024. Available: https://www.cncf.io/reports/cncf-annual-survey-2023/

[3] Kubernetes, "Container Runtime Interface (CRI)," Kubernetes Documentation, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/cri/

[4] D. Crosby et al., "containerd: An Industry-Standard Container Runtime," CNCF Graduated Project Documentation, 2023. Available: https://containerd.io/

[5] M. Heon et al., "CRI-O: Lightweight Container Runtime for Kubernetes," Red Hat, Tech. Rep., 2023.

[6] S. Stanišić, V. Mladenović, T. Ivan, and B. Đorđević, "A Comparative Analysis of Network Policy Implementation in Kubernetes: Leveraging Flannel and Calico for Enhanced Security and Performance," AlfaTech, Dec. 2025. doi: 10.46793/AlfaTech1.2.44S

[7] S. Stanišić, M. Vesković, O. Ristić, and B. Đorđević, "Security Aspects of Container Orchestration in Kubernetes Environments," in Proc. 24th Int. Symp. INFOTEH-JAHORINA, Mar. 2025. doi:10.1109/INFOTEH64129.2025.10959185

[8] S. Stanišić, B. Đorđević, O. Ristić, and T. Ivan, "Performance Optimization of File Systems for Docker Containers," in Proc. Sinteza 2025, Jan. 2025. doi: 10.15308/Sinteza-2025-117-127

[9] Kubernetes, "Dockershim Removal FAQ," Kubernetes Blog, 2022. [Online]. Available: https://kubernetes.io/blog/2022/02/17/dockershim-faq/

[10] Containerd Authors, "containerd Architecture," GitHub, 2026. [Online]. Available: https://github.com/containerd/containerd

[11] CRI-O Authors, "CRI-O - OCI-based implementation of Kubernetes Container Runtime Interface," GitHub, 2026. [Online]. Available: https://github.com/cri-o/cri-o

[12] S. Stanišić and V. Mladenović, "Implementing Network Policies in Kubernetes," in Proc. ALFATECH 25, Feb. 2025. doi: 10.46793/ALFATECHproc25.165S

[13] S. Stanišić and N. Zogović, "A Multi-Objective Approach to Optimizing Cloud Infrastructure with Genetic Algorithms," in Proc. 12th Int. Conf. IcETRAN, Aug. 2025. doi: 10.1109/icetran66854.2025.11114286