

Comparative analysis of selected containerization tools in terms of MCP

Paweł Jan Thusty*, Maciej Pańczyk

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

Model Context Protocol (MCP) extends AI model capabilities in modern software architectures through standardized tool interfaces. In such systems, MCP servers are often deployed as containerized services, making container runtime behaviour a critical factor for startup responsiveness, execution performance, and infrastructure efficiency. Despite extensive research on container runtime performance and growing interest in MCP-based systems, their interaction has received limited research attention. This paper presents an empirical evaluation of container runtimes executing an MCP server under controlled conditions. Metrics including Time To First Successful Response (TTFSR), CPU usage, memory footprint, and p95 latency were collected across multiple scenarios. Statistical analysis was conducted using non-parametric methods with multiple-comparison correction and effect size estimation. The results indicate that cold-start latency differences remain inconclusive, while measurable differences are observed in memory usage and CPU consumption. These findings suggest that runtime selection should be guided by workload characteristics rather than architectural assumptions alone.

Keywords: Model Context Protocol; Artificial Intelligence; containerization; Docker; Podman; containerd

*Corresponding author

Email address: s95596@pollub.edu.pl (P. J. Thusty)

Published under Creative Common License (CC BY 4.0 Int.)

1. Introduction

A fundamental principle of high-quality software systems is the ease with which their functionality can be extended without compromising maintainability or correctness. Modern software architectures continuously more rely on structure of small components where the design quality can influence the behaviour and scalability of the overall system. Virtualization and containerization technologies have been created over many years as valid responses to the need for unified environments and practical trade-offs between isolation strength, performance and resource management. Every new concept added abstraction which addressed specific technical limitation, increasing system complexity, affecting not only time to lead time for changes but also operational costs, including infrastructure management and energy consumption.

The growing integration of artificial intelligence models into the software development lifecycle has further amplified these challenges. Modern AI-driven systems gradually more require handling of operational context, access control, and failover mechanisms. Techniques such as prompt engineering, context engineering, fine-tuning address parts of this problem but remain insufficient for complex multi domain workflows. As a result, a complementary approach has emerged in which model capabilities are extended through the integration with external resources and tools, often leading to agent-based architectures of models, servers, tools, hosts, and clients operating across heterogeneous environments.

This evolution has determined the emergence of standardized protocols such as the Model Context Protocol (MCP), which formalizes structured interactions between models and external tools, further named as

Data Layer. MCP currently supports two transport mechanisms, standard input/output and HTTP streams [1].

While existing studies typically compare container runtimes in terms of usage regular scripts and web applications or focus on MCP from architectural, security, or automation perspectives, there is limited resource on how container runtimes behave when executing MCP servers. In particular, startup responsiveness, resource efficiency, and performance under sustained tool-based interaction remain underexplored, motivating the experimental investigation presented in this paper.

2. Literature review

Despite a substantial body of literature on container runtime performance and rapidly growing patterns of agentic and tool-based AI Systems, empirical studies that mutually examine containerization tool in the context of protocol-driven, tool-based services remain limited.

Containerization has been extensively studied as a lightweight alternative to hypervisor-based virtualization, with work establishing that containers can achieve near-native performance for CPU and memory workloads while introducing overhead primarily through input/output and operating system interactions. Merkel's [2] foundational work on Docker describes the architectural principles underlying container technologies, emphasizing Linux namespaces and control groups as the mechanisms enabling fast startup, efficient resource utilization, and lightweight isolation.

Subsequent empirical studies have focused on quantifying container performance relative to virtual machines and native execution. Felter et al. [3] demonstrate that containers often match or outperform virtual machines in CPU and memory intensive workloads, while highlighting that filesystem layers and network address

translation can introduce substantial overhead if left unconfigured. These findings emphasize the importance of controlling storage and networking settings in performance evaluations and motivate the careful experimental design adopted in later studies.

More recent research has shifted attention toward comparisons among container engines themselves. Baresi et al. [4] present a comprehensive qualitative and quantitative evaluation of multiple container engines across cloud, HPC, and IoT contexts, reporting substantial differences in startup latency and memory footprint between engines. Their results show that while engines such as Docker and Podman exhibit similar startup times, Podman may display higher variance under concurrent startup conditions. Although this work provides strong methodological guidance and highlights engine level differences, it evaluates only synthetic or application benchmarks and does not consider protocol driven or interactive workloads.

Complementing this perspective, Melo et al. [5] evaluate Docker and Podman with a focus on container management task such as image pulling, container instantiation, and teardown. Their results indicate that Podman instantiates small containers faster, whereas Docker performs better for larger images and removal operations. This work demonstrates that container lifecycle overhead is sensitive to image size and execution scenario, an insight directly relevant to ephemeral services. However, the study does not measure application level responsiveness, resource usage during execution, or request-response latency.

Other comparative analyses examine container technologies from a broader virtualization perspective. Silva et al. [6] compare Docker and LXD, showing that both technologies exhibit low steady state overhead and high maturity, particularly on Linux systems. Their results suggest that system containers can offer slightly better stability and consistency than application containers. Nevertheless, the evaluation focuses exclusively on long running throughput benchmarks and does not address startup behaviour or minimal service execution.

Beyond runtime behaviour, several studies emphasize that container performance is strongly influenced by image construction and configuration practices. Ksonitini et al. [7] analyse the evolution of Dockerfiles across hundreds of open-source projects and report that Docker image size and build duration exhibit systematic growth patterns over a project's lifecycle. In particular, they observe that image size increases steadily during early development stages, peaks around the mid-project phase with an average increase of approximately 31%, and later stabilizes with only minor reductions. Similarly, Dockerfile build duration rises throughout development, reaching increases of up to 50% in later stages of the project. These findings highlight that container image characteristics are not static artifacts but evolve over time, and that uncontrolled image growth can significantly affect startup latency and operational costs, thereby confounding runtime performance comparisons if not explicitly controlled.

Host environment characteristics can further influence observed container performance and act as a confounding factor in runtime comparisons. Sobieraj and Kotyński [8] evaluate Docker performance across Linux, Windows, and macOS platforms and show that Docker deployments on non-Linux hosts rely on an additional lightweight virtual machine layer to emulate a Linux environment, which can materially affect CPU, storage input/output, and network performance. Their results demonstrate that the operating system and virtualization path can dominate performance outcomes, with different platforms exhibiting advantages across distinct subsystems.

In parallel with container research, a growing body of work investigates agent-based AI systems and architectures that extend model capabilities through external tools. Preprint by Meyerson et al. [9] was consistent with that long-horizon task reliability can be achieved through extreme decomposition and orchestration of micro-agents, drawing explicit parallels between micro-agent architectures and microservices.

Several applied studies explore structured communication mechanisms for coordinating multi-agent systems in practical domains. For example, Sciammarelli [10] proposes a multi-agent trading architecture that employs a lightweight structured messaging schema referred to as MCP to facilitate decoupled and auditable interactions between agents.

Research on MCP and related frameworks predominantly focuses on architectural design, safety, automation, and human impact. These works examine how standardized tool interfaces enable deterministic and extensible agent behaviour, but they typically abstract away from infrastructure-level considerations such as container runtime performance, startup responsiveness, and resource efficiency.

Taken together, the studies by Baresi et al., Melo et al., and Silva et al. [4–6] provide complementary insights into container engine behaviour, lifecycle costs, and steady-state performance, while Meyerson et al. [9] offer a contrasting architectural perspective on large-scale tool orchestration and reliability, underscoring the need to evaluate not only algorithmic correctness but also the efficiency of the underlying execution environment.

Collectively existing literature provides extensive insights into container runtime performance under traditional workloads and into the architectural properties of tool-based and agentic AI systems. However, these research streams have largely evolved in parallel, with limited cross-analysis at the empirical level. Current container benchmarks do not evaluate protocol-driven, interactive services such as MCP servers, while MCP-related research. Both theoretical and applied does not account for the operational characteristics of container runtimes. In particular, there is limited empirical evidence on how different container runtimes behave when executing MCP servers, where startup responsiveness, resource efficiency, and sustained tool-based interaction are critical. This gap motivates the experimental investigation presented in this work. These observations suggest that the

behaviour of complex, tool-based systems is fundamentally constrained by the performance characteristics of their smallest execution units, underscoring the importance of evaluating container runtimes at level of individual, short-lived services.

3. Scope of work and hypotheses

Main objective of this paper is evaluation of the runtime-level behaviour of container platforms when executing a Model Context Protocol (MCP) server under controlled and reproducible conditions. The scope is restricted to Linux-based, CPU-only workloads and focuses on startup responsiveness (H1), container-level memory utilisation (H2), and CPU usage under sustained workload (H3). Scenarios involving external APIs or remote model calls are excluded from hypothesis testing and treated as integration benchmarks. The following research hypotheses have been formulated:

1. H1: Under equivalent experimental conditions and identical MCP workloads, the mean Time To First Successful Response (TTFSR) statistically differs between containerd and Docker.
2. H2: Under sustained MCP server workloads, the mean container-level memory usage, as measured by cgroup-based working set metrics, statistically differs between containerd and Docker.
3. H3: During sustained MCP server workloads, there is a statistically significant difference in mean CPU utilization among Docker, Podman, and containerd.
4. H4: Distribution of warm tool-call latency (p95) statistically differs across Docker, Podman, and containerd.

The statistical methods used for hypothesis evaluation are described in Section 4.6. The hypotheses are grounded in differences in implementation of container runtimes, especially architecture, resource management, networking and request execution paths.

4. Methods

4.1. Experimental environment

All experiments were executed on a dedicated Linux host equipped with an AMD Ryzen 7 4800U processor, 1.8 GHz, 8 GB RAM, and Ubuntu 24.04 LTS. Storage capacity was sufficient for the benchmark artifacts and was not observed to be a limiting factor.

The system configuration, including software versions and runtime parameters, was recorded automatically prior to each experimental run using a preflight script. A summary of the environment is provided in Table 1.

All containers were executed in rootful mode to ensure consistent cgroup-based resource accounting. The experimental host was reserved exclusively for benchmarking, and background services were minimized to reduce external interference.

The evaluated container runtimes included Docker, Podman, and containerd (*ctr* interface). Hereafter, containerd and *ctr* are used interchangeably.

Table 1: Experimental Environment and Runtime Configuration

Component	Version / Value
Operating System	Ubuntu 24.04.3 LTS
Kernel	6.17.0-14-generic
Architecture	X86_64
cgroup	v2 (unified; cpu, memory, io, pids, cpuset enabled)
Python	3.12.3
Docker	29.1.3
Podman	4.9.3
containerd / ctr	2.2.1
FastMCP	2.14.1

The benchmark workload was implemented using a reference MCP server built on *python:3.12-slim* base image and executed as a non-root user. It exposes an HTTP JSON-RPC endpoint on port 8000 and a Prometheus metrics endpoint on port 9090. A health endpoint (*GET /health -> {"status": "ok"}*) was used for readiness detection.

Four tools were implemented:

- *noop*, performing a near-baseline workload with negligible computational cost,
 - *cpu_task*, performing iterative SHA-256 hashing,
 - *experiment_start*, and *experiment_end*, used for session bookkeeping,
 - *classify_text_http*, issuing an external HTTP request.
- Only *noop* and *cpu_task* were used in the benchmark scenarios described in this study.

An observability stack was deployed and remained active throughout the entire experimental session. Its components and role are summarised in Table 2.

Table 2: Observability and Metrics Infrastructure

Component	Version	Role
Prometheus	2.53.0	Metric scraping and storage
Pushgateway	1.9.0	Receives per-iteration metrics
Grafana	11.1.0	Dashboard visualisation
Cgroup scraper	custom	Polls <i>cpu.stat</i> and <i>memory.current</i> at 1 s intervals
Loki + Promtail	-	Container log aggregation

4.2. Metrics

The following metrics are collected during the experiments:

- Time To First Successful Response (TTFSR): the elapsed time between container start initiation and the first successful readiness response from the MCP server,
- CPU utilization: container-level CPU usage observed during execution,
- Memory usage: container-level memory consumption observed during execution,

- Latency: response latency for request-driven scenarios, reported using percentile statistics where applicable,
- Throughput: number of successful tool invocations per unit time,
- Error rate: proportion of failed readiness probes or tool invocations.

4.3. Test Cases

The benchmark consists of following scenarios which are used to evaluate the MCP server under different runtime conditions:

- S1 – Cold start: The container image is present on the host, and a new container instance is created and started. TTFSR is measured as the elapsed time until the service becomes responsive,
- S2 – Warm single-request: A running container processes a single no-operation tool invocation under low concurrency to establish a warm baseline,
- S3 – Sustained CPU workload: A CPU-bound tool is executed under fixed concurrency and duration to evaluate sustained CPU utilization behavior,
- S4 – Sustained idle: The container remains running without processing requests to measure baseline memory usage,
- S5 – Restart recovery: The container is restarted, and TTFSR is measured after the restart cycle. A single *noop* tool call is issued to verify first-tool responsiveness.

4.4. Suite Parameters

The Table 3 summarises the benchmark suite configuration, including scenario identifiers, planned number of measured runs per runtime (n), and concurrency level. Each scenario targets a specific aspect of runtime behaviour under controlled experimental conditions.

Table 3: Benchmark Suite Configuration

Suite	Scenario	n
S1	cold_start	185
S2	warm_baseline	20
S3	cpu_sustained	14
S4	idle_memory	81
S5	restart_recovery	20

4.5. Procedure

For each runtime and scenario, the benchmark script initiates container execution and records the start trigger time. TTFSR is measured as the elapsed time until the service responds successfully. For sustained workload scenarios, measurements are collected over a fixed execution window after readiness is confirmed. Each scenario is executed multiple times, and results are aggregated for analysis.

Each measured iteration follows a fixed sequence:

1. (S1/S5 only) Pre-run commands are executed (*mcp-drop-caches* and, for *ctr*, *mcp-ctr-gc*).

2. The container start command is issued via a runtime adapter implemented using Python's subprocess module with a list-form argument array (*shell=False*), thereby standardising runner-level invocation across runtimes and avoiding shell-level interpretation of command arguments.
3. The health endpoint is polled at intervals of *interval_ms* until either an HTTP 200 response is observed or a timeout occurs. TTFSR is recorded at the first successful response. Timeout attempts are visible in runner logs, but they do not produce TTFSR records in the result dataset. If the retry limit is exhausted, the measurement slot is excluded from the dataset.
4. A workload phase is applied according to the scenario declaration: concurrent tool calls are generated for *duration_s* seconds (S2/S3), no workload is applied (S4), or a single tool invocation is performed (S5).
5. Upon completion of the workload phase, scenario-specific metrics are collected. For sustained or idle phases, CPU and memory metrics are captured using a cgroup-based scraper. Measurements are exported to Pushgateway for monitoring and aggregation.
6. Finally, the container is stopped and removed, and the resulting data are appended to the run CSV dataset.

4.6. Statistical Design

All measurements were treated as non-Gaussian. Exploratory visualisations of container startup and latency distributions (Figures 1 and 2) indicated right-skewed behaviour and supported the use of non-parametric statistical tests.

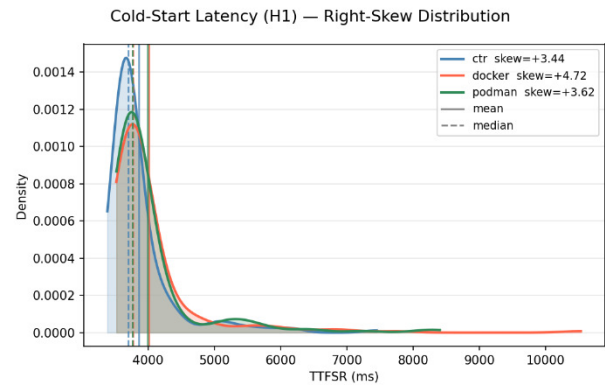


Figure 1: Exploratory distribution of cold-start TTFSR (H1)

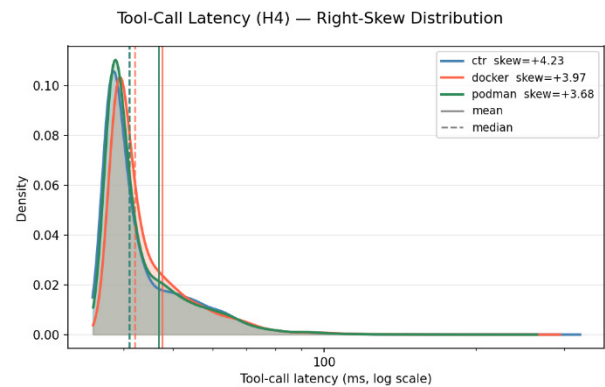


Figure 2: Cold-Exploratory distribution of warm tool-call latency (H4)

Hypothesis testing:

- two-sample comparisons. (H1, H2) were evaluated using the Mann-Whitney U test (two sided),
- multi-sample comparisons (H3, H4) were evaluated using the Kruskal-Wallis H test, followed by Dunn's post-hoc test for pairwise comparisons when the null hypothesis was rejected.

To control the familywise error rate ($\alpha = 0.05$), Holm-Bonferroni correction was applied. Statistical significance was determined using adjusted p-values.

Effect sizes were reported as testing:

- rank-biserial correlation (Mann-Whitney),
- eta-squared (Kruskal-Wallis),
- Cohen's d for comparability.

95% confidence intervals were estimated using t-distribution-based approach. Statistical power was approximated using parametric analogues (t-test and ANOVA), adjusted for asymptotic relative efficiency.

4.7. Failure Handling

Up to two retries were permitted per measurement slot in the event of transient failures. Each retry produced an independent observation. Containers were recreated between runs to ensure independence of observations and to prevent metric carryover effects between measurements.

5. Results

All results were derived from the analysis Jupyter Notebook. Statistical significance was evaluated at $\alpha = 0.05$ with Holm-Bonferroni correction applied where appropriate.

5.1. Cold-start latency (H1)

The hypothesis (H1) that containerd (ctr) cold-start latency differs from Docker was not conclusively supported.

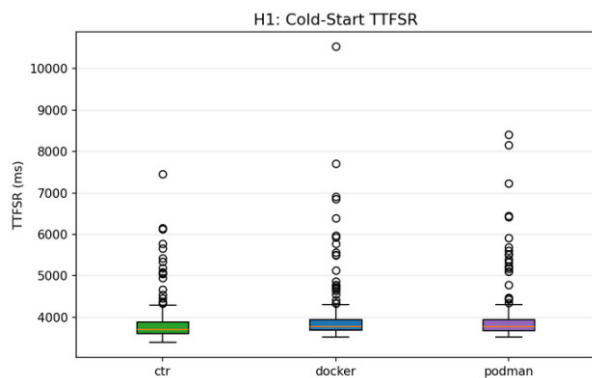


Figure 3: Cold-start TTFSR distribution across container runtimes (H1)

Measured TTFSR values were numerically close across runtimes, with median values of 3700 ms for ctr, 3774 ms for Docker, and 3769 ms for Podman. The Mann-Whitney U test indicated a statistically significant Docker-ctr difference after Holm-Bonferroni correction ($U = 21967.0$, $p_{adj} < 0.0001$). However, the observed

effect was small (Cohen's $d = 0.220$) and statistical power remained below the target threshold.

Inspection of the distributions revealed right-skewed upper tails, and high-latency observations across all runtimes, consistent with cache warm-up, operating system scheduling, or runtime jitter. Sensitivity analysis excluding extreme observations did not materially change the conclusion.

Although ctr showed a slightly lower cold-start TTFSR than Docker, the observed difference was small and insufficiently powered (0.54) for a robust directional claim. H1 was treated as undecidable. No reliable cold-start advantage was established.

5.2. Idle memory footprint (H2)

The hypothesis (H2) that containerd and Docker statistically differ in idle memory usage was confirmed.

Containerd exhibited higher idle memory consumption than Docker. Statistically significant difference was observed using the Mann-Whitney U test ($U = 1991.0$, $p < 0.0001$), with containerd consuming approximately 7.55 MB more memory on average than Docker under idle conditions. The observed effect size was small (Cohen's $d = 0.229$), indicating that the difference was measurable but limited in magnitude.

This result indicates that runtime selection affects the baseline cost of keeping MCP services resident, which is particularly relevant for high-density or high-availability deployments.

Table 4: Summary statistics of idle memory usage (MB) by runtime

	ctr	Docker	Podman
n	81	81	81
Median memory (MB)	287	266	276
Mean memory (MB)	284	277	285

Although H2 formally compared ctr and Docker, Podman is included in Table 4 as descriptive context because it was evaluated under the same idle scenario and proves an additional baseline for interpreting runtime-level memory behaviour.

Overall, Docker exhibited the lowest idle memory, while containerd showed a statistically significant idle memory overhead relative to Docker. Podman remained below containerd by median memory but did not constitute the lowest-memory opinion when mean memory was considered.

5.3. Sustained CPU utilisation (H3)

The hypothesis (H3) that CPU utilisation statistically differs across runtimes under sustained workload was confirmed.

Table 5: CPU usage statistics during sustained MCP workload

	ctr	Docker	Podman
n	13	14	14
Median CPU	10.54 %	9.35 %	10.70 %
Mean CPU	10.52 %	9.24 %	10.53 %

The Kruskal-Wallis test indicated a statistically significant difference in CPU utilisation distributions across

runtimes ($H = 16.183$, $p = 0.000306$, $\eta^2 = 0.4501$), with high estimated statistical power (close to 1.00). Post-hoc Dunn comparisons showed that Docker differed significantly from both ctr ($p_{adj} = 0.0020$) and Podman ($p_{adj} = 0.0009$), while no significant difference was observed between ctr and Podman ($p_{adj} = 0.7897$).

Docker showed the lowest sustained CPU utilisation, with median CPU usage approximately 1.2–1.4 percentage points lower than ctr and Podman.

This finding suggests that runtime overhead persists during steady-state operation and is not limited to startup behaviour.

5.4. Warm concurrent latency (H4)

The hypothesis (H4) that warm p95 tool-call latency differs across Docker, Podman, and containerd was not supported.

Table 6: Summary of p95 tool-call latency under concurrent load

	ctr	Docker	Podman
n	20	20	20
Median p95	71.0 ms	70.8 ms	70.5 ms
Mean p95	72.0 ms	72.5 ms	71.9 ms

Measured p95 latency values were highly similar across all runtimes. The Kruskal-Wallis test did not indicate a significant distributional difference ($H = 0.112$, $p = 0.9453$, $\eta^2 = 0.0029$). Post-hoc comparisons were also non-significant. Given the low achieved power (0.063), this result should be interpreted as insufficient evidence of a measurable latency difference, not as proof of equivalence.

To support interpretation of the p95 metric, Table 7 reports supplementary per-call latency was also summarised computed across all individual tool calls. These summaries are descriptive only. The H4 inference was based on per-attempt p95 values, whereas the pooled per-call statistics were used to characterise the underlying latency distribution.

Table 7: Descriptive per-call latency statistics under concurrent load

	ctr	Docker	Podman
n	12 778	12 590	12 798
Median	40.9 ms	42.1 ms	41.0 ms
Mean	46.7 ms	47.6 ms	46.9 ms
Pooled p95	72.2 ms	72.6 ms	72.2 ms

The per-call summaries in Table 7 are consistent with the H4 result: mean, median, and p95 latency were nearly indistinguishable across runtimes. The difference in completed call counts reflects the fixed 30-second workload window rather than an unequal experimental input.

The expected advantage of host networking in ctr over bridge-based networking in Docker and Podman was not observed, suggesting that networking configuration was not the dominant factor in latency under these conditions.

H4 was not supported. No reliable warm-latency advantage was established for any runtime under concurrent MCP tool-call workloads.

6. Conclusions

This study provides a controlled and reproducible of container runtimes under MCP-style workloads. The results show that runtime performance cannot be described by a single metric or simple rule, and that commonly held assumptions about runtime behaviour do not consistently hold under realistic conditions.

The primary contribution of this work lies in demonstrating that:

- performance differences between runtimes are context-dependent, rather than universal,
- simplified assumptions (e.g. “daemonless runtimes are always faster”) are not consistently supported in practice,
- cold-start latency differences remained inconclusive under the current experimental design,
- clear and measurable differences were observed in memory usage and CPU consumption,
- observed behaviour depends more strongly on the workload scenario rather than on runtime architecture alone.

Taken together, these results emphasise the importance of evaluating container runtimes under realistic and well-defined workloads.

6.1. Practical synthesis

No runtime dominated across all measured dimensions. Instead, each runtime exhibits distinct characteristics:

- Docker: lowest sustained CPU usage and lowest idle memory usage in the measured run, with no significant warm-latency disadvantage, indicating strong suitability for long-running MCP services,
- containerd (ctr): provided the lowest observed cold-start TTFSR and warm latency comparable to the other runtimes, making it relevant for lightweight or latency-sensitive deployment scenarios,
- Podman: balanced behaviour across all measured metrics.

This suggests that runtime selection should be workload-driven, rather than based on architectural preference alone.

At the same time, the differences observed in this study are moderate rather than absolute. In many cases, application-level behaviour and system configuration may have a greater impact than the choice of runtime itself.

6.2. Critical Examination of Testing Methods

The results of this study are subject to several threats to validity. Container runtime performance is sensitive to configuration choices, host environment, and caching effects, which may influence measurements independently of the runtime itself. To mitigate this, all experiments use identical MCP server implementations, equivalent container images, fixed resource limits, and a controlled host environment.

Startup responsiveness is captured using TTFSR, which reflects protocol readiness rather than container start time alone, but may still be influenced

by implementation-specific initialization logic. Resource usage was measured using cgroup-based metrics to ensure comparability across runtimes. However, memory interpretation requires caution because *memory.current* includes file-backed pages and therefore does not isolate anonymous application working-set memory.

Finally, the evaluation covered a limited subset of MCP workload scenarios and a single hardware and operating system configuration. Consequently, the conclusions should be restricted to the tested scenarios, runtime versions, and measurement conditions.

6.3. Closing remarks

The results highlight that container runtime evaluation benefits from a multi-dimensional perspective, where latency, CPU, and memory are considered jointly rather than in isolation. Architectural expectations did not automatically translate into measurable advantages across all scenarios. Instead, runtime behaviour was shaped by the interaction between runtime, architecture, workload pattern, and host-level system dynamics.

From a practical standpoint, the study encourages a more evidence-driven approach to runtime selection, where decisions are informed by measured behaviour under representative workloads.

Future work may extend this analysis by increasing sample sizes for cold-start measurements, evaluating additional workload types, measuring anonymous memory separately from total cgroup memory, and exploring multi-node or distributed deployments.

References

- [1] Model Context Protocol Architecture. Architecture overview, <https://modelcontextprotocol.io/docs/learn/architecture>, [01.01.2026].
- [2] D. Merkel, Docker: lightweight Linux containers for consistent development and deployment, Linux Journal 239 (2014) 2, <https://dl.acm.org/doi/10.5555/2600239.2600241.2600241>.
- [3] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An Updated Performance Comparison of Virtual Machines and Linux Containers, In 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (2015) 171–172, <https://doi.org/10.1109/ISPASS.2015.7095802>.
- [4] L. Baresi, G. Quattrocchi, N. Rasi, A qualitative and quantitative analysis of container engines, Journal of Systems and Software 210 (2024) 111965, <https://doi.org/10.1016/j.jss.2024.111965>.
- [5] P. Melo, L. Gama, J. Dantas, D. Beserra, J. Araujo, Performance Evaluation of Container Management Tasks in OS-Level Virtualization Platforms, In 2023 IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE) (2023) 1–6, <https://doi.org/10.1109/WETICE57085.2023.10477783>.
- [6] D. Silva, J. Rafael, A. Fonte, Toward Optimal Virtualization: An Updated Comparative Analysis of Docker and LXD Container Technologies, Computers 13(4) (2024) 94, <https://doi.org/10.3390/computers13040094>.
- [7] E. Ksontini, M. Mastouri, R. Khalsi, W. Kessentini, Refactoring for Dockerfile Quality: A Dive into Developer Practices and Automation Potential, In 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR) (2025) 788–800, <https://doi.org/10.1109/MSR66628.2025.00116>.
- [8] M. Sobieraj, D. Kotyński, Docker Performance Evaluation across Operating Systems, Applied Sciences 14 (2024) 1–16, <https://doi.org/10.3390/app14156672>.
- [9] E. Meyerson, G. Paolo, R. Dailey, H. Shahrzad, O. Francon, C. F. Hayes, X. Qiu, B. Hodjat, R. Miikkulainen, Solving a Million-Step LLM Task with Zero Errors, arXiv preprint arXiv:2511.09030 (2025), <https://doi.org/10.48550/arXiv.2511.09030>.
- [10] J. Sciammarelli, Multi-Agent Trading Systems Applied in the Brazilian Finance Market, International Journal of Advanced Research 13 (2025) 523–527, <https://doi.org/10.21474/IJAR01/22135>.