Memory Efficient WebAssembly Containers

Matthijs Jansen
VU Amsterdam
Amsterdam, The Netherlands
m.s.jansen@vu.nl

Maciej Kozub
VU Amsterdam
Amsterdam, The Netherlands
m.p.kozub@student.vu.nl

Alexandru Iosup
VU Amsterdam
Amsterdam, The Netherlands
a.iosup@vu.nl

Daniele Bonetta
VU Amsterdam
Amsterdam, The Netherlands
d.bonetta@vu.nl

***Abstract*—WebAssembly (Wasm) is a portable, high-performance binary instruction format originally designed for web browsers. It is rapidly gaining traction in server-side applications, including containerized environments orchestrated by Kubernetes. However, the performance impact of a widespread WebAssembly adoption in containerized environments remains unclear. Solutions relying on suboptimal WebAssembly runtimes may increase performance and memory overhead, especially in high-density deployment settings.**

In this paper, we explore the impact of WebAssembly runtimes on containerized application deployment in Kubernetes. Through an in-depth analysis of existing Wasm container runtimes, we identify inefficiencies in memory usage and startup times, limiting Wasm's viability in large-scale deployments. We propose a new integration of the lightweight WebAssembly Micro Runtime (WAMR) into the crun container runtime to resolve the identified inefficiencies of Wasm containers. Benchmark evaluations demonstrate that our WAMR integration reduces memory usage by 11% to 78% per container compared to existing Wasm runtimes while outperforming 4 of 6 benchmarked runtimes in container startup time. Furthermore, our integration reduces memory usage compared to Python containers by at least 16% and startup time by 3% to 18%. Our findings show that Wasm containers are a competitive alternative to traditional non-Wasm container solutions, especially in dense container deployments, and highlight the importance of runtime optimization in cloud-native environments. Our work is open-source and available at https://github.com/atlarge-research/continuum/tree/wasm.

***Index Terms*—WebAssembly, Containers, Kubernetes, Cloud-native, Memory efficiency, Performance.**

I. Introduction

The landscape of cloud computing has undergone a significant transformation in recent years. Containerized applications have become widely adopted, largely due to the need for scalable, flexible, and maintainable software deployments [1]. Containers are a cost-effective yet high-performance technique for isolating application resources. Applications can operate in separate contexts while sharing the host operating system kernel, filesystem, and resources [2]. Orchestration platforms like Kubernetes automate deployment, scaling, and management of containerized environments. The open-source nature of Kubernetes and its cloud-native approach have revolutionized the computing landscape in the last decade and made it a popular choice for orchestrating containerized workloads at scale [3], [4]. Industry reports show over 60% of companies have adopted Kubernetes, a figure projected to surpass 90% by 2027 [5]. Other reports show that 5.6 million developers use Kubernetes and global sales of containerized solutions have reached 1.2 billion dollars in 2022 [6].

Along with the advantages brought by containerization, there is also a need for efficient container execution as running applications in containers creates an overhead that translates to higher resource usage and energy consumption [7]. The surge in demand for cloud computing services has further exacerbated data centers' energy consumption and carbon footprint, making energy costs one of the top operational expenses [8]. Moreover, the high velocity of change in the number of running containers in large-scale deployment environments leads to spikes in resource utilization [9]. Cluster providers need to accommodate more hardware to maintain a high availability and scalability of services during peak hours, further increasing operational costs [10].

Despite its advantages, containerization introduces resource inefficiencies, particularly in high-density deployments where memory overhead becomes a critical concern, such as serverless computing [11]. WebAssembly [12] (Wasm), originally designed for Web browsers, has gained attention as a lightweight, portable, and secure runtime for executing server-side applications. Using Wasm, developers can create applications that are securely sandboxed, highly portable across diverse hardware and software environments, and capable of near-native execution speeds. However, integrating Wasm into containerized environments poses challenges. Existing Wasm-enabled container runtimes, such as Wasmtime [13] and WasmEdge [14], often exhibit higher memory overhead than traditional container runtimes, limiting their scalability.

This paper addresses the overhead and scalability challenges of Wasm containers by integrating the WebAssembly Micro Runtime (WAMR) [15] into the *crun* container runtime [16] and evaluating its impact on memory efficiency and performance in Kubernetes workloads. To motivate our integration approach, we first analyze the container execution process of Kubernetes and define a methodology for integrating new

WebAssembly runtimes into Kubernetes. We conduct a comprehensive set of experiments to compare the performance of multiple WebAssembly runtimes. Our results reveal that different Wasm runtimes exhibit highly varied performance characteristics, demonstrating that runtime selection is crucial in optimizing containerized applications. The contributions of this work are as follows:

1) **Integration**: We systematically analyze the current support of WebAssembly in Kubernetes and synthesize a methodology on integrating WebAssembly runtimes into the Kubernetes container orchestrator (Section III). Based on our analysis of WebAssembly and container runtimes, we propose a novel integration of the WAMR WebAssembly runtime into the crun container runtime to lower container memory overhead compared to existing WebAssembly integrations.
2) **Benchmarking**: We conduct comprehensive benchmarks evaluating memory usage, startup performance, and scalability of different Wasm runtimes in containerized Kubernetes environments (Section IV). Our novel WAMR-crun integration shows superior performance, with faster container startup compared to 4 of 6 benchmarked WebAssembly runtimes and 11% to 78% lower container memory consumption. Moreover, we outperform traditional Python containers in memory (at least 16%) and startup performance (3% to 18%).
3) **Discussion**: Based on our results, we provide insights into the design and implementation of Wasm container runtimes for cloud-native deployments (Section IV).

By highlighting the performance and resource disparities between runtimes, our work underscores the importance of carefully evaluating and optimizing Wasm engines for Kubernetes workloads. Our work is open-source and available at https://github.com/atlarge-research/continuum/tree/wasm.

II. Background: Containers and Kubernetes

Containers are an essential technology for modern software development and deployment, encapsulating applications and their dependencies in a portable, isolated unit. Unlike OS-level virtual machines, containers share the host operating system kernel, resulting in reduced resource usage and faster startup times. Popular container runtimes, such as Docker and crun, manage the lifecycle of containers, including creation, execution, and teardown.

Kubernetes [17] extends the utility of containers by orchestrating their deployment across clusters of machines. It abstracts infrastructure complexities, enabling developers to focus on application logic. Kubernetes employs a modular architecture with a control plane that manages worker nodes. The control plane deploys a centralized API server, scheduler, and database for decision-making, and a daemon per node to communicate with a container runtime through a standardized container runtime interface (CRI) and operating system. Multiple container runtimes support this CRI, giving Kubernetes flexibility in choosing runtime implementations that best meet specific performance or security requirements.

WebAssembly (Wasm) is a stack-based virtual machine initially designed to execute code in web browsers. It compiles high-level languages like Rust, C, and Go into a compact binary format that can run consistently across diverse platforms. Wasm modules operate in a secure, sandboxed environment, making them ideal for executing untrusted code. The sandboxed environment is more lightweight than traditional virtual machines and containers, allowing Wasm-sandboxed applications to execute at near-native performance [18]. The WebAssembly System Interface (WASI) [19] extends Wasm's capabilities, allowing modules to perform system-level operations like file access and networking.

The integration of Wasm into containerized environments has garnered interest because of its potential to reduce container image sizes, improve startup performance, and improve security [20], [21]. However, we find that Wasm support in the Kubernetes container orchestrator is complex and ill-understood, resulting in a broad performance and resource use heterogeneity between Wasm-enabled container runtimes. Furthermore, we find that current Wasm containers face limitations in memory efficiency compared to Wasm modules, a critical factor in large-scale Kubernetes deployments. Addressing these inefficiencies is crucial for a more extensive adoption of Wasm in cloud-native computing.

III. Integrating a Lighter WebAssembly Runtime into crun

The primary objective of this paper is to assess the impact of using lighter WebAssembly runtimes on container memory consumption in Kubernetes. By integrating a lighter runtime, we aim to reduce the memory overhead of Wasm containers while maintaining compatibility with existing Kubernetes workflows. In this section, we first analyze current support of WebAssembly in Kubernetes (Section III-A) to guide us in selecting a new lightweight Wasm and container runtime that can improve memory overhead over existing approaches. Next, we synthesize a methodology for integrating such a lightweight Wasm runtime into Kubernetes (Section III-B) and use it to integrate the WebAssembly Micro Runtime (WAMR) into the *crun* container runtime (Section III-C).

A. Current support of WebAssembly in Kubernetes

We illustrate the existing integration of WebAssembly container within Kubernetes in Figure 1, following the Wasm guidelines set by the Cloud Native Computing Foundation [22]. At the top, Kubernetes serves as the container management platform, orchestrating containerized applications' deployment, scaling, and operation. It interfaces through the container runtime interface (CRI) with a high-level container runtime, such as containerd, CRI-O, or Docker Engine, that manages container lifecycle operation and relies on a low-level container runtime to execute these operations through the operating system. Kubernetes uses containerd by default.

Containerd deploys daemon processes on worker nodes to communicate with the lower-level container runtimes. These

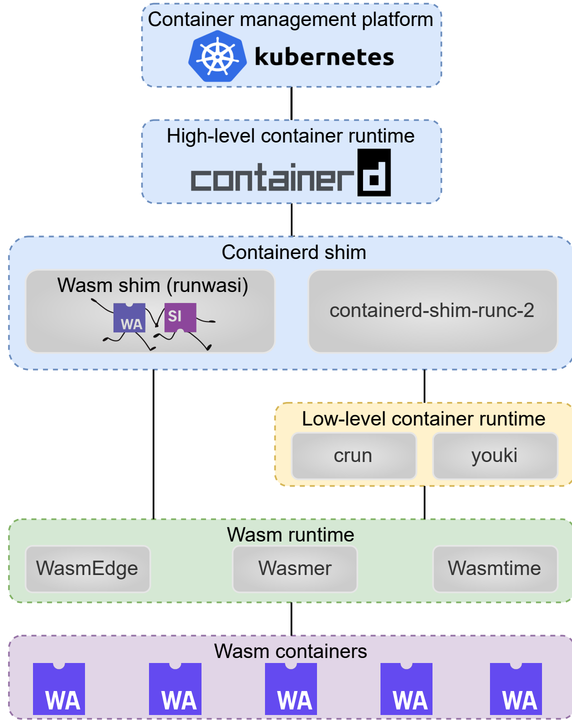

Fig. 1. Current WebAssembly support in Kubernetes.

daemons utilize shim processes, lightweight intermediary processes running between the containerd daemon and lower-level container runtimes, to manage container instances. By using a shim, containerd can ensure that container processes are decoupled from the containerd daemon, which allows the daemon to be restarted or upgraded without affecting running containers, enhancing the system's reliability.

Containerd uses the lower-level container runtime runC by default to execute OCI-compliant (Open Container Initiative) containers via the *containerd-shim-runc-v2* shim. The Open Container Initiative is a set of standards defining how to operate a container, enforced by Kubernetes' CRI. We show in Figure 1 shows that containerd can manage other low-level container runtimes besides runC, such as crun and youki, as they are OCI-compliant. Crun and youki support Wasm containers through the WasmEdge, Wasmer, and Wasmtime Wasm runtimes. Alternatively, the containerd runwasi project delivers a set of shims that facilitate the execution of WebAssembly containers by bridging containerd directly with Wasm runtimes, bypassing low-level container runtimes.

B. How to add a new Wasm runtime to Kubernetes?

The complex and hierarchical nature of Kubernetes' container ecosystem makes adding support for a new Wasm runtime to an existing container runtime, and reasoning what runtime to support and how, non-trivial [23]. For container runtimes, Kubernetes allows developers to choose a low-level container runtime (e.g., runC, crun, youki, etc.) and independently exchange high-level container runtimes (e.g., containerd, CRI-O, Docker Engine, etc.), as long as they conform to the Container Runtime Interface (CRI) [24]. The myriad of possible container runtime combinations poses optimization opportunities but requires a thorough analysis and benchmarking to support reasoning about optimizations, which currently does not exist. Additionally, the decision on the Wasm runtime to integrate is non-trivial, as many runtimes exist with different performance characteristics, properties, and different support for specific features such as the emerging WebAssembly System Interface [19] (see Section II). In this work, we focus on integrating a Wasm runtime with the highest potential in terms of memory savings. To this end, we have conducted extensive testing of existing Wasm support, which we present in Section IV, and preliminary testing of new Wasm runtimes to assess their memory consumption. Specifically, we address the following two design questions:

1) Which Wasm runtime should we choose between Wasmer, Wasmtime, WasmEdge, and Wamr?
2) Should we integrate the Wasm runtime into the low-level crun or youki container runtimes, or directly into containerd via runwasi?

Based on preliminary testing and existing research into Wasm [25], we have decided to use crun as a container runtime and WAMR as the WebAssembly runtime. In Section IV, we provide experiments showing how integrating the two components leads to significant performance improvements. We selected crun as the container runtime due to its lightweight nature and performance efficiency, which are crucial for optimizing resource utilization while ensuring fast startup times. Additionally, crun has demonstrated superior support for the Open Container Initiative (OCI) specifications, which aligns with our goals of maintaining industry standards and compatibility across various environments. On the other hand, WAMR was chosen as the WebAssembly runtime due to its minimal footprint and effective execution capabilities on constrained devices. WAMR is optimized for small code size, low latency, and high efficiency, making it an ideal choice for deploying WebAssembly applications in scenarios where resource constraints and performance are paramount.

C. Integrating WAMR into crun

Integrating WAMR into crun involves several key steps to address the inherent differences between traditional container runtimes and WebAssembly environments. These steps are non-trivial, requiring changes to ensure compatibility with crun's lightweight architecture while maintaining adherence to Kubernetes' APIs. At a high level, the core of our integration has been structured around the following aspects:

1) **Dynamic Library Loading:** The WAMR shared library is dynamically loaded at runtime, ensuring minimal memory footprint when Wasm containers are not used.
2) **WASI Argument Handling:** The integration supports passing environment variables, runtime arguments, and pre-opened directories to Wasm modules, ensuring compatibility with existing containerized workflows.

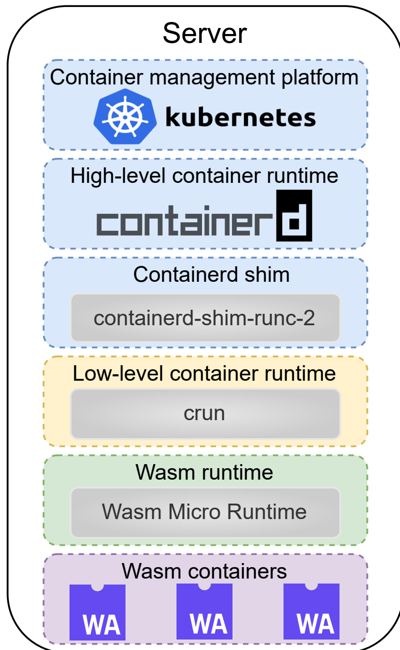

Fig. 2. Overview of our WAMR integration into crun.

3) **Sandboxed Execution:** Each Wasm module is executed in a secure, isolated environment, leveraging WAMR's compact runtime and Kubernetes' namespace isolation.

The modified crun runtime retains full compatibility with Kubernetes. Kubernetes pods, the scheduling unit of Kubernetes, can seamlessly run traditional and Wasm-based containers, enabling hybrid deployments without additional infrastructure changes. An overview of the general architecture of our integration of WAMS into crun is depicted in Figure 2. We extend the Kubernetes cluster configuration to evaluate our integration at scale, now supporting up to 500 pods per node (each pod can run many containers at once). This extension requires modifications to the kubelet configuration and the cluster's networking setup, ensuring that high-density workloads can be executed without bottlenecks.

IV. Impact of Different WebAssembly Runtimes on Kubernetes

The performance profiles of different WebAssembly runtimes remain insufficiently explored and understood, particularly in the context of containerized workloads managed by Kubernetes. To address this gap, we conduct a series of experiments to evaluate how various WebAssembly runtimes perform in the context of different container and Kubernetes pod densities. These experiments focus on performance metrics such as memory usage, startup latency, and resource consumption, providing an overview of the general runtime behavior of WebAssembly-based containers. In addition to evaluating existing Wasm-based containers, we also perform targeted experiments to assess our integration of the WebAssembly Micro Runtime into the crun container runtime. Our experiments demonstrate how the WAMR integration positively impacts Kubernetes workloads, particularly in terms of memory efficiency and startup performance, and highlight its advantages over existing solutions. By analyzing the performance of different Wasm-based containers, including our WAMR-crun integration, we aim to provide insights into runtime efficiency and identify opportunities for optimization and future enhancements.

TABLE I
Software stack for the evaluation.

Software	Version	Software	Version
Linux	5.4.0-187-generic	WAMR	2.1.0
Kubernetes	1.27.0	WasmEdge	0.14.0
containerd	1.1.12	Wasmer	4.3.5
runC	1.6.31	Wasmtime	23.0.1

TABLE II
Experiments overview. Experiments deploy 10 to 400 containers concurrently, with 1 container per pod.

Section	Metric	Container runtime	Language runtime
§ V-B	Memory	crun	WAMR, WasmEdge, Wasmer, Wasmtime
§ V-C	Memory	crun, containerd	WAMR, WasmEdge, Wasmer, Wasmtime
§ V-D	Memory	crun, runC	WAMR, Python
§ V-E	Latency	crun, runC, containerd	WAMR, WasmEdge, Wasmer, Wasmtime, Python

A. Experimental Setup

We execute our experiments on the Continuum framework [26], which deploys a Kubernetes cluster on an Intel(R) Xeon(R) Silver 4210R CPU with 256GB of RAM and 20 cores running Ubuntu 20.04.3 LTS. We show an overview of the used software in Table II. We have modified containerd, runc, and WAMR to achieve our WAMR integration. In all experiments, we execute a minimal C application corresponding to a very small microservice. Using such a small microservice makes memory and startup performance dominated by the WebAssembly runtime we want to evaluate rather than the actual microservice being executed. We discuss the impact of different applications in Sections IV-D and IV-F. We report memory use per container as an average of the 10 to 400 concurrently deployed containers. The deviation in memory use is negligible at less than 0.1 MB per container.

B. Memory Overhead Compared to Wasm containers in crun

We measure the memory footprint of running Wasm containers in a Kubernetes cluster on a system level through the Linux *free* command and on a resource management level through Kubernetes' metrics server. We show the memory usage per container as reported by the metrics server in Figure 3. The vertical axis shows the memory used in megabytes, where lower values are preferred. The horizontal axis indicates the runtime configuration used to execute the containers. We evaluated four configurations: (i) In red, our new implementation of embedded WebAssembly Micro Runtime in crun; (ii) crun with Wasmtime; (iii) crun with Wasmer; (iv) crun with WasmEdge. The measurements for each runtime were made with deployment densities of 10, 100, and 400 containers per node, with one identical Wasm container per pod. The results

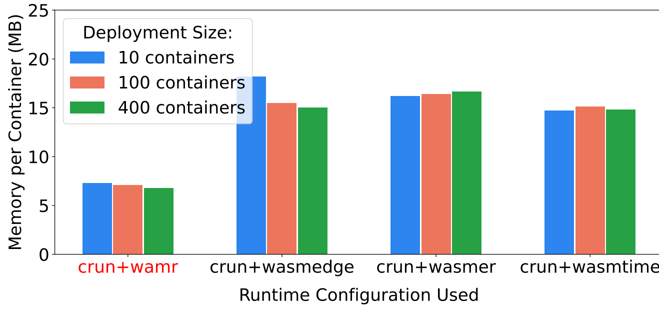

Fig. 3. Average memory usage per container for different Wasm runtimes in crun, measured by Kubernetes. Our work's results are labeled in red.

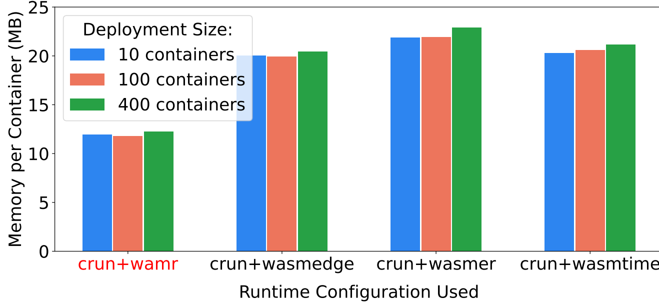

Fig. 4. Average memory usage per container for different Wasm runtimes in crun, measured by the OS.

for each deployment size are indicated by the separate bars for each runtime. Our implementation, in red, outperforms the other three existing Webassembly integrations in crun by at least 50.34% for any deployment density.

Figure 4 reports the measurements of memory used per container for the same experiments but collected using the Linux free command. Compared to the Kubernetes metrics server results, we notice a significant difference in nominal values of used memory, with free reporting higher memory usage in all scenarios, and up to 42% more memory used. This difference is expected as the free command reports the system's overall memory usage, including buffers, system caches, and processes other than those related to the Kubernetes cluster. Meanwhile, the Kubernetes metrics server focuses on resources used by the workloads scheduled on the node.

Figures 3 and 4 compare our work with all other Wasm runtimes supported by the low-level container runtime crun. We can derive that our implementation uses at least 50.34% (reported by the metrics server) and 40.0 % (reported by the free command) less memory to execute Wasm containers than any other Wasm runtime currently supported in crun. We also observe that the memory overhead per container does not vary significantly between different deployment sizes. This finding indicates that our implementation provides proper scaling performance.

C. Memory Overhead Compared to runwasi

We also benchmark our solution against the runwasi-delivered high-level runtimes that support Wasm containers directly from containerd. Such comparison allows us to better understand the WebAssembly supporting technologies cur-

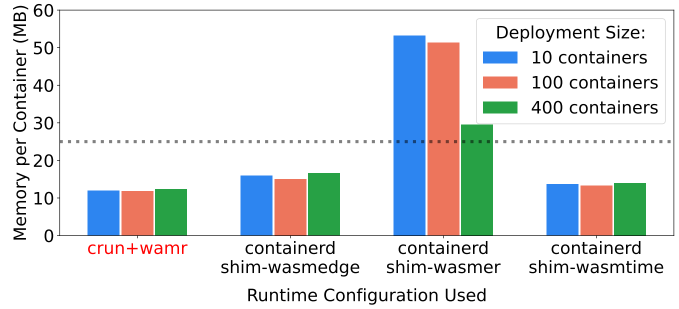

Fig. 5. Average memory usage per container for different Wasm shims, measured by the OS.

rently available and how this work positions itself among them. We replicate the setup from Figure 4 with the free command and report the results of the runwasi-integrated runtime shims in Figure 5. Our implementation, in red, reports the same measurements across the figures.

Our implementation introduces the lowest memory usage per container compared to all available runwasi shims, regardless of the deployment size. Compared to the second-best Wasm runtime in our benchmark, containerd-shim-wasmtime, we reduce memory usage by at least 10.87%. However, the difference is smaller than when we compared our work against Wasm runtimes embedded in crun, where we reduced memory usage by at least 40.0% compared to the second-best runtime, crun-wasmedge. Across the crun and runwasi WebAssembly runtimes, containerd-shim-wasmer reports the worst memory overhead, with our implementation reducing memory usage by 77.53% over all three deployment densities, as reported by the free command.

D. Memory Overhead Compared to Non-Wasm Containers

We want to gain insight into how our work could be positioned among available containerization technologies regarding the memory overhead of running a container on Kubernetes. Thus, as a reference, we also compare our new implementation of the WAMR runtime embedded into the crun container runtime against the standard Python container image. This comparison does not provide direct insights into the distribution and execution of WebAssembly code inside an OCI container or the state of WebAssembly support in Kubernetes. However, we think this comparison helps us understand the maturity of WebAssembly support in containerized environments such as Kubernetes, the visibility of further development, and challenges that WebAssembly still faces.

Figure 6 and 7 present the measurements of memory used per container, collected using the Kubernetes metrics server and the Linux free command, respectively. We compare our implementation, marked in red, with well-established container runtimes without Wasm support. Specifically, we compare our work to a standard Python container deployed with the crun and runC container runtimes. By including crun, we can gain valuable insight into how crun performs without the need to call an underlying WebAssembly runtime and how it compares with our implementation. We also select

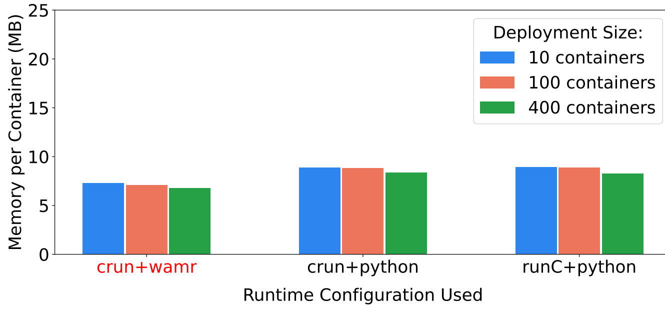

Fig. 6. Average memory usage per container by our work compared with Python containers, measured by Kubernetes.

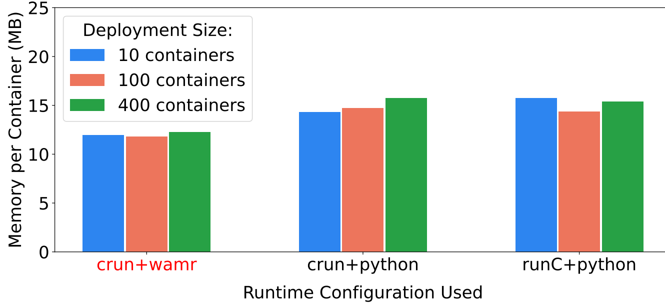

Fig. 7. Average memory usage per container by our work compared with Python containers, measured by the OS.

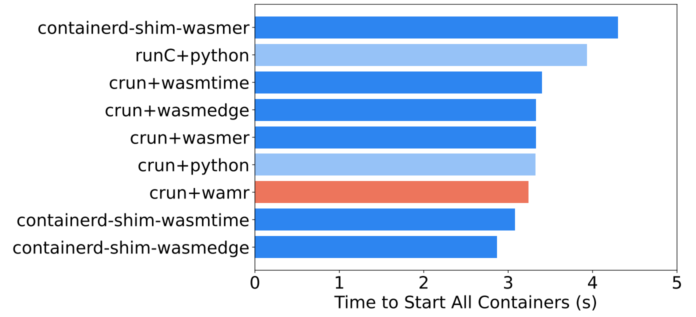

Fig. 8. Time to start 10 concurrent containers' workload executions for different container runtimes. Results for non-Wasm containers are marked in a lighter blue color; our work result is marked in orange.

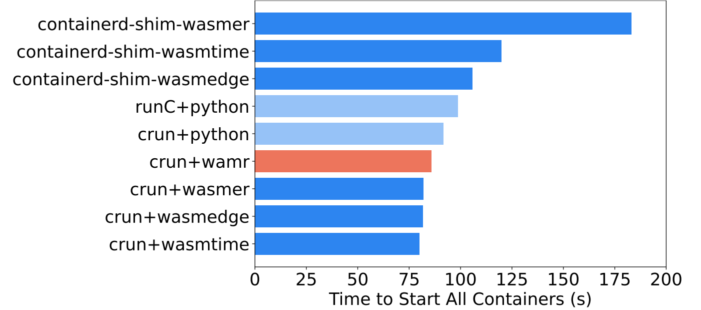

Fig. 9. Time to start 400 concurrent containers' workload executions for different container runtimes.

runC because it is the default low-level container runtime for containerd and Kubernetes. Therefore, we can compare and challenge our work and the WebAssembly containers against the performance of the out-of-the-box Kubernetes cluster with Python containers.

The Kubernetes metrics server measurements (Figure 6) show that our Wasm integration uses at least 17.98% less memory compared to crun and Python, and at least 18.15% less memory compared to runC and Python. Our Wasm integration is the only Wasm runtime that uses less memory than these Python container runtimes. Our integration uses 21.07% less memory than the second-most memory-efficient Wasm runtime according to the metrics server measurements, containerd-shim-wasmtime. For the evaluations with the Linux free command in Figure 7, our implementation uses at least 16.38% and 17.87% less memory than the crun and runC python runtimes. This result shows that Wasm is a competitive alternative to established container runtimes and programming languages for Kubernetes deployments. The second-most memory efficient Wasm runtime, containerd-shim-wasmtime, is now at least 4.66% more efficient than the Python containers, being the only Wasm runtime besides our new integration to outperform Python containers in memory usage.

E. Startup Performance

Figure 8 presents the measurements of time needed to deploy ten containers at once. We start the measurements on deploying pods to Kubernetes and finish the measurements when our sample application starts executing in the last deployed container and pod. The horizontal axis in the figure represents the deployment time elapsed in seconds. Lower values are preferred because they indicate a higher startup performance. The vertical axis indicates the container runtime used to execute the container workload. The result obtained by our work, that is, by crun with an embedded WAMR runtime, is marked in red. The Python containers and runtimes we compared in Section IV-D are marked in light blue. All Wasm runtimes bar for our new integration are indicated in blue.

From figure 8, we can derive that our work does not introduce performance degradation for small deployments of 10 containers. The WebAssembly Micro Runtime embedded in crun executes all containers' WASM modules in under 3.24 seconds, which is below the average across all tested runtimes. The results indicate that containerd-shim-wasmedge and containerd-shim-wasmtime runtimes have the best startup performance for small deployments, taking up to 11.45 % less time to start Wasm modules than our implementation. However, our work performs at least 2.66 % better than any other WebAssembly runtime integrated into crun. Moreover, our integration starts WebAssembly containers faster than the crun and runC runtimes start Python containers.

We also benchmark the startup performance for the larger deployment of 400 containers to better evaluate the behavior of our implementation under heavy loads and test its scalability performance. Figure 9 depicts the time needed to start all 400 containers and containers. We find that our implementation outperforms the containerd-shim-wasmtime and containerd-shim-wasmedge runtimes, which outperformed our implementation when deploying 10 containers at once. For a deployment of 400 containers, our integration took respectively 18.82 % and 28.38 % less time to start all Wasm modules than the

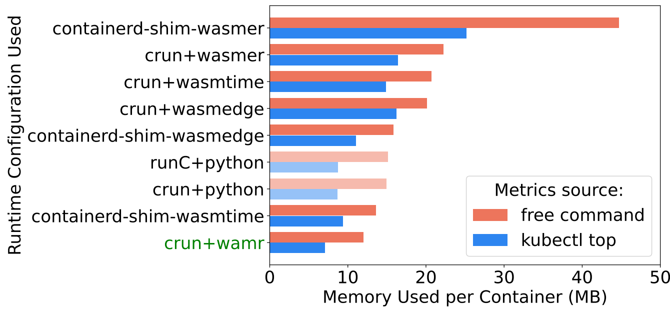

Fig. 10. Memory usage per container by our work (labeled in green) compared with other container runtimes, averaged over all deployment sizes. Results for non-Wasm containers are marked in lighter colors.

containerd-shim-wasmedge and containerd- shim-wasmtime runtimes. However, our implementation shows lower startup performance for larger deployments than currently available WebAssembly runtimes supported in crun. When deploying 400 containers with WebAssembly containers, our work took 6.93% more time to start the Wasm modules execution than the most performant crun-Wasmtime runtime integration. Nonetheless, our approach still presents better startup performance when starting 400 Wasm containers than both crun and runC runtimes with Python containers.

F. Discussion and Overview

Our experiments showed that our implementation delivers on the goal of lowering the memory footprint of Wasm containers. Considering the results obtained from the system using the free command, our solution uses at least 40.0% less memory to run one pod with a Wasm container than any other Wasm runtime integrated with crun. At the same time, our implementation uses at least 10.87% and up to 77.53% less memory to run a pod compared to runwasi shims with WebAssembly support. Finally, our integration uses at least 16.38% less memory than Python containers with the crun and runC container runtimes, the latter being Kubernetes' default container runtime. We provide a comprehensive overview of memory use across all test runtimes in Figure 10. We also found that our implementation delivers startup performance comparable to alternative runtimes regardless of the deployment size, ranging from 10 to 400 pods and containers. This finding shows that our memory optimization did not sacrifice startup performance and that Wasm containers are a competitive alternative to traditional containers such as Python on Kubernetes.

V. Related Work

Research on WebAssembly's server-side applications has highlighted its potential for cloud-native computing. Projects like WasmEdge, Wasmer, and Wasmtime have focused on enhancing runtime performance. However, these solutions often prioritize execution speed over memory efficiency. This work addresses this gap by integrating WAMR into crun, emphasizing memory use.

From the best of our understanding, we are the first to comprehensively benchmark the resource use and performance of WebAssembly runtimes integrated into container runtimes with WebAssembly support. Closest to our work, Sangeeta et al. [27] and Wang [28] provide systematic evaluations of WASM runtimes. However, these works benchmark WASM runtimes in isolation, not integrated into container runtimes, which we show significantly changes the memory and performance characteristics of WebAssembly applications.

Below, we highlight several works in the broader research field of WebAssembly that are adjacent to our research. Jiang et al. [29] investigate performance issues in server-side WebAssembly runtimes. They introduce WarpDiff, a differential testing approach to detect performance issues by comparing execution times across different Wasm runtimes. The study applied WarpDiff to five popular Wasm runtimes, tested with 123 cases from the LLVM test suite, and identified seven performance issues. These issues, confirmed by developers, highlight areas needing optimization in Wasm runtimes. Wiegratz [30] concludes that WebAssembly can complement Linux containers in cloud computing, offering specific benefits in security and efficiency. The work suggests that future developments could further enhance the integration of WebAssembly in cloud-native applications, potentially leading to broader adoption alongside traditional container technologies.

Containerd version 2.0 introduced the Sandbox API [31]. This new API aims to support various sandbox types, including those based on virtual machines and WebAssembly, providing a more flexible way to manage different container environments simultaneously. Kuasar [32], a Cloud Native Computing Foundation sandbox project, aims to provide a secure, efficient, and flexible container runtime for modern cloud-native applications using this new API. Kuasar includes a Wasm sandbox, which allows containers to be launched within a WebAssembly runtime like WasmEdge or Wasmtime with more runtimes planned for future releases. Although the Sandbox API is in the experimental phase, it could provide significant real-world improvements to interoperability and support for external sandboxing due to containerd being the default Kubernetes container runtime. Projects implementing this API, such as Kuasar, might gain rapid traction in the future, requiring a new iteration of our benchmarking and integration work.

VI. Conclusion

In this paper, we presented the WebAssembly Micro Runtime (WAMR) integration into the crun container runtime, demonstrating significant improvements in memory efficiency and deployment performance. By reducing memory overhead and improving startup times, our work helps positioning Wasm containers as a viable alternative for Kubernetes workloads. Furthermore, our experiments with other runtimes highlight that different WebAssembly engines can lead to significant performance variations in Kubernetes workloads, emphasizing the critical role of runtime selection and optimization in achieving optimal efficiency in high-density cloud workloads.

Future research will explore advanced runtime optimizations, multi-tenant scenarios, and broader adoption of Wasm in cloud-native ecosystems.

Acknowledgment

This work was supported by a National Growth Fund through the Dutch 6G flagship project "Future Network Services" and NWO TOP project OffSense. This research was partly suported by the EU Horizon Graph Massivizer and the EU MSCA Cloudstars projects.

References

[1] S. Hassan, R. Bahsoon, and R. Buyya, "Systematic scalability analysis for microservices granularity adaptation design decisions," *Softw. Pract. Exp.*, vol. 52, no. 6, pp. 1378–1401, 2022. [Online]. Available: https://doi.org/10.1002/spe.3069

[2] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *14th IEEE International Symposium on Network Computing and Applications, NCA 2015, Cambridge, MA, USA, September 28-30, 2015*, D. R. Avresky and Y. Busnel, Eds. IEEE Computer Society, 2015, pp. 27–34. [Online]. Available: https://doi.org/10.1109/NCA.2015.49

[3] Z. Rejiba and J. Chamanara, "Custom scheduling in kubernetes: A survey on common problems and solution approaches," *ACM Comput. Surv.*, vol. 55, no. 7, pp. 151:1–151:37, 2023. [Online]. Available: https://doi.org/10.1145/3544788

[4] B. C. Senel, M. Mouchet, J. Cappos, T. Friedman, O. Fourmaux, and R. McGeer, "Multitenant containers as a service (caas) for clouds and edge clouds," *IEEE Access*, vol. 11, pp. 144 574–144 601, 2023. [Online]. Available: https://doi.org/10.1109/ACCESS.2023.3344486

[5] Datahub Analytics Team, "Kubernetes: Adoption and market trends," https://datahubanalytics.com/kubernetes-adoption-and-market-trends/, 2024, accessed 2025-01-15.

[6] Edge Delta, "Kubernetes adoption statistics: Unveiling global trends," *Edge Delta Blog*, May 2024, accessed: 2025-01-10. [Online]. Available: https://edgedelta.com/company/blog/kubernetes-adoption-statistics

[7] S. Park and H. Bahn, "Performance analysis of container effect in deep learning workloads and implications," *Applied Sciences*, vol. 13, no. 21, 2023. [Online]. Available: https://www.mdpi.com/2076-3417/13/21/11654

[8] P. E. Mergos, "Seismic design of reinforced concrete frames for minimum embodied co2 emissions," *Energy and Buildings*, vol. 162, pp. 177–186, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378778817327664

[9] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload analysis and demand prediction of enterprise data center applications," in *IEEE 10th International Symposium on Workload Characterization, IISWC 2007, Boston, MA, USA, 27-29 September, 2007*. IEEE Computer Society, 2007, pp. 171–180. [Online]. Available: https://doi.org/10.1109/IISWC.2007.4362193

[10] B. Erdenebat, B. Bud, and T. Kozsik, "Challenges in service discovery for microservices deployed in a kubernetes cluster – a case study," *Infocommunications Journal*, vol. Special Issue on Applied Informatics, pp. 69–75, 2023. [Online]. Available: https://www.infocommunications.hu/Spec_5_11

[11] C. Xu, Y. Liu, Z. Li, Q. Chen, H. Zhao, D. Zeng, Q. Peng, X. Wu, H. Zhao, S. Fu, and M. Guo, "Faasmem: Improving memory efficiency of serverless computing with memory pool architecture," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, Eds. ACM, 2024, pp. 331–348. [Online]. Available: https://doi.org/10.1145/3620666.3651355

[12] WebAssembly, "Webassembly," https://webassembly.org/, 2025, accessed: 2025-01-21.

[13] Wasmtime, "Wasmtime: A fast and secure webassembly runtime," https://wasmtime.dev/, 2025, accessed: 2025-01-16.

[14] WasmEdge, "Wasmedge: High-performance webassembly runtime for cloud native, edge, and decentralized applications," https://wasmedge.org/, 2025, accessed: 2025-01-21.

[15] Bytecode Alliance, "Webassembly micro runtime," https://bytecodealliance.github.io/wamr.dev/, 2025, accessed: 2025-01-21.

[16] containers, "crun: A fast and lightweight fully featured oci runtime and c library for running containers," https://github.com/containers/crun, 2025, accessed: 2025-01-20.

[17] Kubernetes, "Kubernetes," https://kubernetes.io/, 2025, accessed: 2025-01-21.

[18] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: a serverless-first, light-weight wasm runtime for the edge," in *Proceedings of the 21st International Middleware Conference*. Association for Computing Machinery, 2020, pp. 265–279. [Online]. Available: https://doi.org/10.1145/3423211.3425680

[19] WebAssembly, "Webassembly system interface," https://github.com/WebAssembly/WASI, 2025, accessed: 2025-01-21.

[20] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski, "Webassembly modules as lightweight containers for liquid iot applications," in *Web Engineering - 21st International Conference, ICWE 2021, Biarritz, France, May 18-21, 2021, Proceedings*, ser. Lecture Notes in Computer Science, M. Brambilla, R. Chbeir, F. Frasincar, and I. Manolescu, Eds., vol. 12706. Springer, 2021, pp. 328–336. [Online]. Available: https://doi.org/10.1007/978-3-030-74296-6_25

[21] M. Sebrechts, T. Ramlot, S. Borny, T. Goethals, B. Volckaert, and F. D. Turck, "Adapting kubernetes controllers to the edge: on-demand control planes using wasm and WASI," in *11th IEEE International Conference on Cloud Networking, CloudNet 2022, Paris, France, November 7-10, 2022*. IEEE, 2022, pp. 195–202. [Online]. Available: https://doi.org/10.1109/CloudNet55617.2022.9978884

[22] Seven Cheng, "Webassembly on kubernetes: The practice guide part 02," March 2024, accessed: 2025-01-11.

[23] N. Zhou, Y. Georgiou, M. Pospieszny, L. Zhong, H. Zhou, C. Niethammer, B. Pejak, O. Marko, and D. Hoppe, "Container orchestration on HPC systems through kubernetes," *J. Cloud Comput.*, vol. 10, no. 1, p. 16, 2021. [Online]. Available: https://doi.org/10.1186/s13677-021-00231-z

[24] E. Truyen, H. Xie, and W. Joosen, "Vendor-agnostic reconfiguration of kubernetes clusters in cloud federations," *Future Internet*, vol. 15, no. 2, p. 63, 2023. [Online]. Available: https://doi.org/10.3390/fi15020063

[25] Y. Zhang, M. Liu, H. Wang, Y. Ma, G. Huang, and X. Liu, "Research on webassembly runtimes: A survey," *CoRR*, vol. abs/2404.12621, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.12621

[26] M. Jansen, L. Wagner, A. Trivedi, and A. Iosup, "Continuum: Automate infrastructure deployment and benchmarking in the compute continuum," in *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering, ICPE 2023, Coimbra, Portugal, April 15-19, 2023*, M. Vieira, V. Cardellini, A. D. Marco, and P. Tuma, Eds. ACM, 2023, pp. 181–188. [Online]. Available: https://doi.org/10.1145/3578245.3584936

[27] S. Kakati and M. Brorsson, "A cross-architecture evaluation of webassembly in the cloud-edge continuum," in *24th IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2024, Philadelphia, PA, USA, May 6-9, 2024*. IEEE, 2024, pp. 337–346. [Online]. Available: https://doi.org/10.1109/CCGrid59990.2024.00046

[28] W. Wang, "How far we've come - A characterization study of standalone webassembly runtimes," in *IEEE International Symposium on Workload Characterization, IISWC 2022, Austin, TX, USA, November 6-8, 2022*. IEEE, 2022, pp. 228–241. [Online]. Available: https://doi.org/10.1109/IISWC55918.2022.00028

[29] S. Jiang, R. Zeng, Z. Rao, J. Gu, Y. Zhou, and M. R. Lyu, "Revealing performance issues in server-side webassembly runtimes via differential testing," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 661–672. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00088

[30] J. A. Wiegratz, "Comparing security and efficiency of webassembly and linux containers in kubernetes cloud computing," *CoRR*, vol. abs/2411.03344, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2411.03344

[31] containerd, "Proposal: Sandbox api," https://github.com/containerd/containerd/issues/4131, 2021, accessed: 2025-01-18.

[32] Kuasar Project, "Wasm sandboxer," https://kuasar.io/docs/architecture/wasm-sandboxer/, 2023, accessed: 2025-01-16.