The Final Frontier of Orchestration: Bringing Kubernetes to On-Field and Embedded Devices Beyond Cloud-to-Edge Continuum

Pallav Kumar Deb, Jorge Carola, Susmit Shegokar, Wolfgang Forstmeier
Siemens Technology and Services Private Limited,
Germany
e-mail: (pallav.deb, jorge.carola, susmit.shegokar, wolfgang.forstmeier)@siemens.com

***Abstract*—The resource demands of cloud-native orchestration tools make it challenging to include last-mile devices, particularly embedded and on-field systems, in Cloud-to-Edge continuum deployments. This represents the final missing piece in achieving truly end-to-end orchestration across federated environments. This paper introduces a federated architecture that extends Kubernetes (K8s)-native orchestration beyond the Cloud-to-Edge continuum, reaching deeply embedded on-field devices such as actuators powered by single-board processors commonly found on industrial floors and in smart buildings. While K8s allows custom shims by leveraging WebAssembly (WASM) runtimes, it is not yet ready to be used on the constrained on-field devices due to its desired memory footprint. This paper introduces a splitted shim, built on the WASM Micro Runtime (WAMR). It decouples the runtime into a lightweight client-side agent and a proxy shim that can be deployed either on an edge node or in the cloud as a service, depending on deployment requirements. This proxy mediates communication with the K8s control plane, enabling orchestration without requiring Open Container Initiative (OCI) bundle support on the constrained device. The architecture is OS- and platform-independent, supports heterogeneous environments, and drastically reduces the resource footprint, enabling scalable, cloud-native orchestration from centralized cloud infrastructure to edge gateways and minimalistic field devices.**

Keywords-cloud–edge–field continuum; orchestration; kubernetes; webassembly (WASM); constrained devices

I. INTRODUCTION

Workload migration is a critical enabler in modern distributed computing environments to seamlessly move applications across heterogeneous infrastructures for optimized performance, resilience, and resource utilization. With the advent of Docker and its salient features, industries have widely adopted containerized architectures. Consequently, platforms such as Kubernetes (K8s) have become the de-facto standard for orchestration and managing workloads at scale. It offers powerful and clean abstractions for deployment, scheduling, monitoring, and service management. Its level of simplification, consistency, and automation has led to undisputed adoption across cloud, edge, and hybrid computing deployments.

K8s depends on key components, such as the Application Programming Interface (API) server, etcd (data store), scheduler, controller manager, kubelet (node agent), and kube-proxy (networking). Together, these components make K8s reliant on resource-capable devices, which confines its suitability to only relatively powerful edge devices and not those operating at the last mile, referred to hereafter as on-field devices. These devices are typical in industrial and field deployments. However, due to their tiny CPUs, limited memory, and ruggedized hardware, running the standard K8s stack is a challenge. For instance, in popular use cases such as environmental sensing, motion and inertial monitoring, biometrics and health data acquisition, distance and proximity measurement, touch and interaction sensing, and similar scenarios, STM-series microcontrollers are the primary units responsible for collecting, processing, and transmitting data. They typically have flash memory (up to 4 MB for high-end series) for code storage and Static Random-Access Memory (SRAM) (up to 1.5 MB) for data handling, which makes them unsuitable for running K8s components. While there are lighter alternatives to K8s available, they are still not ready to be hosted on constrained devices. Table I highlights some of the popular K8s distributions and their hardware requirements [1]. As a result, many real-world systems end up excluding these constrained devices from orchestration entirely, managing them instead with custom scripts or proprietary solutions that do not integrate well with modern cloud-native workflows, eventually limiting interoperability and scalability.

TABLE I. HARDWARE REQUIREMENTS FOR KUBERNETES DISTRIBUTIONS

Environment	Minimum RAM	Minimum CPU	Disk Space
K8s	4 GB	2 cores	20 GB
K3s	512MB	1 core	200MB
Minikube	2GB	2 cores	20GB
Kind	1GB	1 core	5GB
MicroK8s	540MB	1 core	300MB

In this work, we aim to challenge the long-standing assumption that last-mile devices must remain excluded from the K8s ecosystem and isolated from cloud-native capabilities. To address this gap, we propose an approach that allows resource-constrained devices to be onboarded into K8s using off-the-shelf solutions. For brownfield deployments, our method requires only minimal southbound changes, allowing these constrained devices to appear directly on K8s dashboards without requiring any northbound modifications. In our design, the control plane remains agnostic to the underlying limitations of these devices, treating them as standard nodes within the cluster. This enables new opportunities for workload migration, dynamic reconfiguration, and adaptive control from cloud to edge and all the way to on-field devices.

Use Case: Smart-building sensors, such as smoke detectors run on constrained devices. They stay idle most of the time and cannot join orchestration frameworks due to reasons mentioned earlier. Currently, workloads are manually mapped one-to-one.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

The proposed solution enables simple, cloud-native workload orchestration without changing existing models.

A. Motivation

With the advent of WebAssembly System Interface (WASI), WebAssembly (WASM) workloads run outside of the web browser and directly on devices using POSIX-like interfaces [2]. This enhances portability, strict isolation, near-native speed, and cross-language composability. Containerd [0] has started supporting docker-based WASM images by introducing runtimes to their *shims* in contrast to Linux-based shims [3]. A containerd shim is a lightweight daemon that manages the container's lifecycle. This has allowed K8s to also support WASM-based workload orchestration. WebAssembly Micro Runtime (WAMR) is a popular choice for running WASM workloads on constrained devices. However, the containerd shim coupled with the WAMR runtime is ~10 MB, which is not suitable for constrained devices (refer Section I). In addition to the runtime, the workloads have Open Container Initiative (OCI) relevant files included in the images, which further demands space. Such memory requirement alone restricts installation of the K8s components, which is the main motivation of our work. We propose ***Federated Shim*** as a method to onboard constrained device by splitting the shim into 2 components. A proxy component resides on an edge device (or optionally on the cloud for non air gapped devices) and handles K8s operations. The executor resides on the constrained device which is responsible for running the workload. Details about the strategy are available in Section III.

B. Contributions

In this work, we split the containerd shim into 2 components for making them more compatible for constrained devices. This makes them suitable for onboarding to K8s in a cloud-native fashion. Towards this, we make the following contributions:

- ***Federated shim architecture***: We introduce a two-part WASM shim that offloads container-runtime responsibilities to an edge-side proxy while leaving only a lightweight execution agent on the constrained device. This eliminates the need for OCI bundles or containerd on the field device.
- ***Platform-independence***: We reduce the WAMR footprint to ~30 KB through custom builds and Ahead-of-Time (AOT) compilation, enabling devices with only a few hundred kilobytes of RAM to run K8s-orchestrated workloads.
- ***Seamless deployment***: We enable constrained devices to appear as regular K8s nodes using standard dashboards and tooling, requiring no northbound modifications, making it suitable for brownfield deployments.
- ***Status reporting and observability***: We implement a message-driven mechanism that enables sharing logs, health status, and execution results to the K8s control plane, ensuring observability despite device limitations.
- ***Modular, plug-and-play components***: We design a fully modular system. The communication module, execution runtime, proxy layer, and onboarding components are independent and replaceable without affecting the rest of the stack, ensuring portability across heterogeneous hardware and simplifying integration into diverse industrial deployments.

The remainder of this paper is structured as follows. In Section II, we review related work and existing approaches. Section III presents the proposed federated shim architecture. Section IV outlines the experimental setup, and Section V discusses the key observations and performance results. Finally, Section VI concludes the paper and highlights future work.

II. Related work | Methods

The literature frequently cites the vision of a unified compute continuum stretching from massive cloud clusters down to tiny embedded devices, which consistently breaks down at the far edge. K8s has earned its place as the dominant orchestration platform for cloud-native workloads, offering declarative scheduling, self-healing controllers, and a mature industrial ecosystem [4]. The problem is that its architecture was never designed with microcontroller unit (MCU) in mind. A Class 2 MCU typically offers 256 KB of RAM and 256 KB of Flash. Even a stripped-down Kubelet agent consumes hundreds of megabytes at steady state, which is not suitable for constrained hardwares [6][7].

The community has responded with progressively lighter distributions. Goethals *et al.* [6] showed with FLEDGE that a Virtual Kubelet bridge could bring K8s scheduling to low-resource Linux edge nodes like the Raspberry Pi, side-stepping the full control plane while keeping standard kubectl workflows intact. K3s took a different angle, collapsing every control-plane component into a single binary below 100 MB and substantially reducing memory consumption relative to upstream K8s, landing at a practical minimum of around 512 MB RAM on a combined server–agent node [12]. KubeEdge pushed the agent footprint lower still, to on the order of 70–100 MB depending on configuration, by splitting responsibility between a cloud-side Edge Controller and a lean EdgeCore daemon on the device, and adding Message Queuing Telemetry Transport (MQTT)-based IoT messaging through its built-in Event Bus [7]. Taken together, these works prove that the K8s control plane can be cleanly decoupled from its execution environment, a principle that Federated Shim deliberately exploits, but carries all the way down to the MCU.

On the runtime side, the WASM ecosystem has made equally important progress. Wallentowitz *et al.* [13] systematically benchmarked the major WASM runtimes against embedded targets and concluded that WAMR and Wasm3 are the two most viable candidates for microcontroller deployment. WAMR offers the richer feature set AOT and Just in Time (JIT) compilation. Wasm3 trades those features for a smaller interpreter footprint. Mislav *et al.* [14] validated this further with empirical evaluation across the Raspberry Pi Pico, ESP32-C6, and nRF5340, finding that both runtimes deliver acceptable cross-platform portability and execution speed even under strict resource budgets, suggesting that WASM is approaching practical viability for production IoT scenarios and not merely a research curiosity. Comparatively, there are approaches for packaging WASM modules inside standard OCI images to

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

TABLE II. COMPARISON WITH EXISTING SOLUTIONS.

Methods	K8s	WASM Runtime	Workload Migration	OSI Support	OS Support	Constrained Devices	Minimum Hardware	Messaging/ Network
Vaño *et al.* [4]	✓	Partial	✓	✓	Linux	Partial	Edge-class	–
Feather [5]	✓	Wasmtime	–	✓	Linux	✓	∼86 MB RAM	Kubernetes API
FLEDGE [6]	✓	–	✓	✓	Linux	✗	∼60 MB RAM*	OpenVPN Overlay
KubeEdge [7]	✓	–	✓	✓	Linux	✗	70 MB (agent) / 256 MB (device)	MQTT (EventBus)
Tinto *et al.* [8]	–	✓	✓	–	Heterogeneous	✓	Embedded-level	Bytecode Serialization
wasmCloud [9]	✓	Wasmtime	–	✓	Linux / Bare	✓	Embedded nodes	NATS
Krustlet [10]	✓	Wasmtime	–	Partial	Linux	✗	Std PC / Edge	Kubelet API
Ocre [11]	–	✓	–	Partial	RTOS (Zephyr-class)	✓	Embedded-class	REST / Overlay
Federated Shim	✓	**WAMR**	✓	✓	**None (RTOS/Bare)**	✓	**10 KB RAM / MCU**	**MQTT**

*Approximate value; exact minimum hardware requirement not explicitly stated in the paper. Krustlet is no longer actively maintained [10].

maintain registry compatibility [3]. However, they reintroduce the abstraction layers of container runtime, namespace isolation, and image-layer unpacking, that WASM was originally chosen to eliminate.

A meaningful gap persists between what the literature calls lightweight and what a deeply constrained device actually needs. Most solutions that claim a small footprint still assume a Linux kernel and tens of megabytes of RAM [6]. Vaño *et al.* [4] highlighted in their survey that stateful migration mechanisms across the edge cloud continuum are frequently incompatible with OCI registries, which cuts them off from standard cloud-native tooling and forces operators to maintain separate pipelines. Device onboarding is an equally neglected dimension. Lacalle *et al.* [4] observed that most edge frameworks rely on manual provisioning steps that do not scale gracefully to fleets of volatile or intermittently connected hardware. Projects such as Ocre [11] have tried to address the hardware side of this problem directly, bringing container-like semantics to RTOS-class devices through a WASM execution layer, a REST management API, and support for hardware as modest as an Arm Cortex-M3. However, as Goethals *et al.* [6] would recognize, a runtime without an orchestration plane is only half a solution. Ocre does not natively integrate with K8s and provides no mechanism for stateful migration between nodes.

Synthesis: The Federated Shim for embedded devices is motivated directly by the gap that each of these works leaves open. We take the Virtual Kubelet abstraction that Goethals *et al.* [6] demonstrated at the Linux edge and extend it all the way to bare-metal microcontrollers, combining it with MQTT-based zero-touch onboarding so that a physical device registers itself as a standard virtual node without any manual configuration step. A federated shim layer then routes scheduled workloads through containerd into the WAMR runtime [13], which is the same runtime that Wallentowitz *et al.* [13] benchmarked as the most capable embedded WASM engine enabling OCI-compatible, stateful WASM on hardware with only tens of kilobytes of RAM. The result is a single orchestration fabric that finally reaches the on-field devices rather than stopping at the edge, and does so without abandoning the standard K8s APIs that the rest of the stack already depends on. We highlight the key differences of the existing solutions against the proposed federated shim in Table II.

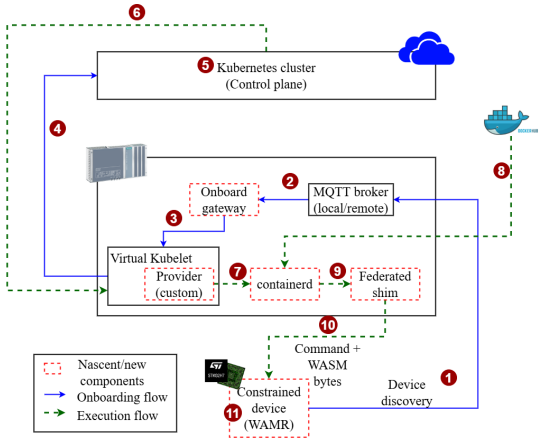

Figure 1. Information flow of the Federated Shim.

III. SYSTEM ARCHITECTURE

We introduce a modular split in the coventional containerd WASM shim to support deployment across heterogeneous environments (refer Figure 1). The edge device serves as a proxy layer, hosting the task service that interfaces with containerd. This service manages lifecycle events and creates WASM instances. Upon instance creation, the edge device forwards execution requests to a constrained device, which hosts the WASM runtime. According to original design, the runtime is invoked through the Youki library, a Rust-based container runtime, enabling secure and efficient execution of WASM workloads. This separation allows the edge device to handle orchestration and communication overhead, while the constrained device focuses solely on lightweight execution.

We adopt WAMR as our choice of runtime as it is tailored for environments with limited memory and compute capacity. While the default WAMR shim build is around 9.2 MB, our target devices have only 500 KB of RAM, requiring significant optimization. A key architectural decision was to eliminate OCI components from the constrained device. These components, typically used for container lifecycle and image management, introduce unnecessary overhead in minimal environments. By removing them, we not only simplify the runtime footprint but also eliminate the dependency on the Rust-based Youki library. Consequently, we focus solely on WAMR, which is written in C. The removal of these primary components does not affect functionality as we introduce agents on both the edge and

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

constrained devices. Compared to the OCI components, the designed agents have significantly low footprint. We further reduce the WAMR runtime size to ~30 KB using custom build flags and AOT compilation. Further, by leveraging WAMR's modular and portable library design, we ensure platform independence and consistent behavior across diverse operating systems without requiring OS-specific adaptations.

The flow of the proposed deployment is as follows:

1) K8s control plane sends control commands to containerd.
   a) Based on deployment strategy, the control plane may be hosted on the edge or the cloud.
2) Containerd forwards to proxy manager.
   a) We use Virtual Kubelet to enable proxy.
3) Proxy manager interacts with the constrained device using a communication module.
4) Communications between the edge device consists of the command along with the WASM bytes.
5) Receiver in the constrained device passes the command and WASM bytes to an execution manager.
6) Execution manager initiates the WAMR runtime and executes the workload.
   a) In the case of multiple workloads, threading is applied as the constrained devices are typically single core processor boards.
7) Based on device type and design requirements, logs of the executions are either stored in the local file system or transferred to the edge device.
8) Monitoring and health status are also sent to the edge device.

The information flow consists of 2 modes: (i) Device onboarding and (ii) Workload execution. As shown in Figure 1, the information flows for each work as follows:

Device onboarding: 1. Constrained device sends a self-discovery message to the edge device. It leverages an MQTT broker (hosted locally/remotely) for sharing messages. 2. The broker forwards the message to an onboarding gateway. 3. The gateway spins up a virtual kubelet, which acts as proxy for the constrained device. 4. Virtual kubelet registers the constrained device on the control plane. 5. K8s control plane registers the device and mandates the use the Federated Shim.

Workload execution: 6. K8s control plane assigns a workload/pod to the constrained device, which is passed to Virtual Kubelet. 7. Virtual kubelet's custom provider triggers containerd. 8. Containerd fetches the pod-specific OCI bundles from the virtual kubelet. 9. Containerd then calls the federated shim interface for the constrained device. 10. Lifecycle commands and WASM bytes are sent to the constrained device. 11. Constrained device executes the workload.

IV. Experiment Setup

In this Section, we present our experiment setup, which mimics real industrial cloud–edge–field environments. We use a Raspberry Pi 4 Model B as the constrained on-field device and a Siemens SIMATIC IPC427E [15] as the edge node. We host the K8s control plane in AWS cloud. The Raspberry Pi 4B is an ARM-Cortex-A72-based microcontroller board with 4 cores, running at 1.8 GHz and 4 GB RAM. It supports Ethernet, dual-band Wi-Fi, and Bluetooth connectivity. While this is not our intended target constrained device, we show consumption metrics in Section V, which proves the suitability of running the proposed solution on the former. The Siemens IPC 427E, in contrast, is an industrial-grade edge system and has sufficient resources to host the proxy components like containerd, Virtual Kubelet, and the proposed federated proxy layer. We choose addition of 2 numbers as our WASM evaluation workload which returns back the sum. Our bias towards this workload is due to its simplicity. It helps us in keeping the computation time negligible (as WASM optimization is beyond the scope of this work) and we primarily focus on the federated shim metrics with ease. Additionally, this workload covers key runtime operations like argument passing across the proxy to on-field device, execution inside a sandboxed module, and retrieval.

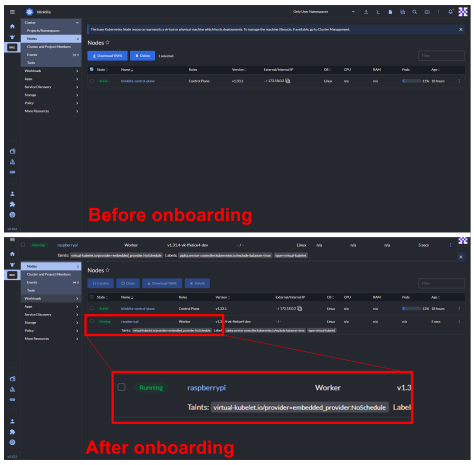

Figure 2. Onboarding the constrained device to K8s (Rancher dashboard).

V. Observations

In this section, we present our observations while deploying the proposed Federated Shim. Since the proxy component can be deployed either on capable edge devices or on the cloud, we focus primarily on the constrained devices. They represent the most critical bottleneck in extending K8s orchestration to last-mile deployments.

A. Device Onboarding

We choose Rancher [16] as our K8s management platform. It is open-source, enterprise-grade, and enables deployment, management, and security for containerized applications across diverse environments. It also simplifies multi-cluster operations by providing a single centralized management console. We validate the onboarding workflow of the proposed solution by monitoring the available nodes in the cluster displayed in the Rancher dashboard. Prior to onboarding, the node list is empty (refer Figure 2 top). Once the constrained device makes the onboarding call, the device appears as a fully registered node (refer Figure 2 bottom). This demonstrates that the proposed Federated Shim onboards the constrained device to the K8s control plane without requiring any northbound modifications.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

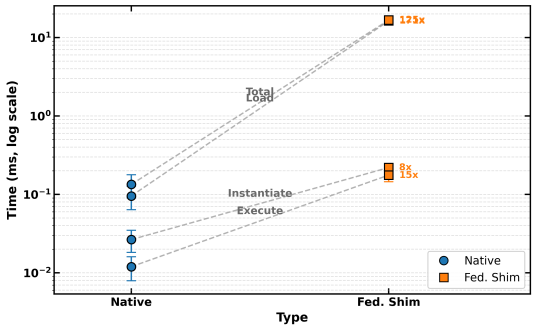

Figure 3. Time consumption in different phases of execution.

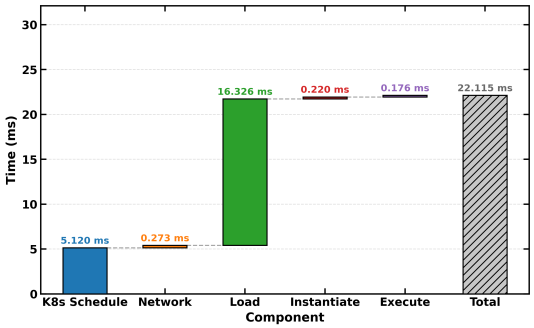

Figure 4. End-to-End Time Consumption.

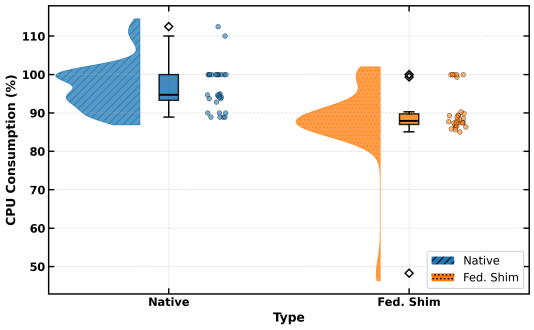

Figure 5. CPU Consumption.

B. Execution Time

In Figure 3, we present the execution times in log scale for better representation. We observe that the total time required for executing the workload by the federated shim is $125x$ compared to native WASM (on average, 16.72 ms and 0.134 ms, respectively). This raises concerns and requires a deep dive. Consequently, we breakdown the time into 3 phases. The *Load* phase reads the WASM binary file from storage into RAM and parses its structure. *Instantiate* phase takes the loaded module and allocates its linear memory, initializes global variables, and wires up all import/export bindings. *Execute* phase hands the fully initialized WASM module directly to the CPU interpreter (WAMR in this case) and runs the actual program logic. We observe that the load phase is the most expensive of all for the federated shim. We attribute this behaviour to requiring disk input/output, deserializing, and initializing WASM. While the instantiation and execution overheads are relatively contained, the federated shim induces overhead when compared to native phases by $8x$ and $15x$, respectively. This observation has been consistent in our experiments suggesting that the federated shim incurs per-invocation cost.

Compared to native execution, the device is advocating MQTT polling and other operations (both OS and shim) while also making sure that the workload executions happen. The differences in the instantiate and execution phases may be occuring due to context switching. However, the higher overhead of loading all components reflects additional marshalling and inter-layer communication costs of the shim rather than fundamental computational inefficiency, which can be improved with modifications.

C. End-to-End Time Consumption

To give a perspective of the scheduling, networking, and data transfer overheads, we decompose the end-to-end invocation latency into 5 components (refer Figure 4). We observe that the total time required to schedule and get results is ~22 ms, of which the actual workload execution time is ~16 ms (refer Section V-B). The K8s scheduling is another major contributor to the time, which is almost 23%. We attribute this to the operations in the control plane for dispatching the workload through the proxy shim, which is using gRPC protocol. This overhead is typical to K8s is independent of the proposed solution and would uniformly apply to any other container runtime shims. On the other hand, the transit time is only 1.2%, demonstrating that the MQTT protocol adds negligible latency. However, we do plan to replace this with protocols that are more suitable for constrained and intermittent networks like nanoIP, CoAP, or others.

We infer from the breakdown the total latency is dominated by the workload execution phases and K8s scheduling, implying that the communication protocols are not a limiting factor. Since the time due to K8s scheduling is a cluster-level byproduct, the workload execution phases contributing ~75% of the time need attention and will be our focus for improvement.

D. CPU Consumption

In Figure 5, we present the CPU consumption during our experiments. Interestingly, we observe that the proposed federated shim, on average, consumes less CPU than native. On the other hand, we observed in Figure 3 longer in wall clock time during execution. With InterQuartile Range (IQR) of 6.69%, the native distribution is tight and uniform, with occasional more than 100% due to compute bursts. The Federated Shim's IQR is even narrower (2.77%), implying that most runs are stable around 88% consumption. The outlier of 48.3% inflates its standard deviation. We conclude that while the runs are consistent, a subset of the executions experience severe CPU under-utilization, which may be caused by prolonged blocking on read/write operations. These non-deterministic scheduling delays yields the CPU entirely.

We infer that the federated shim's overhead is input/output bound, rather than compute-bound, making the IPC communi-

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

TABLE III. OVERHEAD SUMMARY

Phase	Time (×)	Mem. (ΔKB)	CPU (Δ%)
Load	171×	+188	+1.5
Instantiate	8×	+84	+0.5
Execute	15×	+8	-7.6

cation channel (or deployment strategy) as our primary target for improvement and not the WASM runtime.

E. Overhead Summary

We previously observed that the federated shim introduces overhead in execution time. We take a closer look at the memory consumption and then highlight the collective overheads in Table III. The memory consumption follows a similar trend as the execution time in the Load phase by $+188$ KB, we attribute this to the per-invocation IPC channel setup and module binary transfer to the worker process. The instantiation and execution add smaller overheads ($+84$ KB and $+8$ KB, respectively). This may be due to the cross-process linear memory allocation.

We infer that while the execution times and memory allocations are more for the federated shim, the CPU utilization follows an opposite trend. As most clock time is spent blocking on IPC reads rather than active computation (refer previous sections), we reaffirm the IPC channel and deployment strategy as our primary optimization target.

VI. CONCLUSION AND FUTURE WORK

In this work, we addressed the inclusion of embedded and resource-constrained on-field devices in K8s managed ecosystems, which is final missing piece for cloud-edge-on-field orchestration. Existing K8s stack, inclusive of WASM shim support cannot run directly on such single-board industrial microcontrollers due to their resource demands and dependency on OCI bundle support. To bridge this gap, we split the shim architecture and built on the WebAssembly Micro Runtime (WAMR) runtime. This decouples the K8s shim into a lightweight client-side agent, and deployed on the constrained device. The proxy shim is deployed on an edge node or can be deployed as a cloud service. It mediates all interaction with the K8s control plane. Additionally, with the use of strategic techniques, we achieve OS independence and the proposed solution works wherever WAMR is installed, increasing the range of device. Consequently, the device remains agnostic to the orchestration complexity. Through real-world experimentation, we showed the feasibility of the proposed solution and its hardware consumption.

In the future, we plan to extend this work as a derivative of the observations in Section V. We observed that the high execution times and memory consumption are a result of heightened input/output operations and IPC operations, in contrast to compute and MQTT protocol. This necessitates the improvement in the split shim and deployment strategies. We will also be extending this solution to support more (WASM and non-WASM) runtimes and communication protocols.

REFERENCES

[1] A. Iyer, "The Ultimate Guide to Lightweight Kubernetes Environments", Accessed: 2026-02-10, Signadot, May 2025. [Online]. Available: https://www.signadot.com/articles/the-ultimate-guide-to-lightweight-kubernetes-environments

[2] W3C WebAssembly Community Group, "WASI: The WebAssembly System Interface", Accessed: 2026-02-10, WASI Subgroup, W3C WebAssembly Community Group, 2025. [Online]. Available: https://wasi.dev/

[0] containerd authors, *Containerd: An industry-standard container runtime with an emphasis on simplicity, robustness and portability*, Accessed: 2026-03-13, 2026.

[3] S. Cheng, "WebAssembly on Kubernetes: from containers to Wasm (part 01)", Accessed: 2026-02-10, Cloud Native Computing Foundation, Mar. 2024. [Online]. Available: https://www.cncf.io/blog/2024/03/12/webassembly-on-kubernetes-from-containers-to-wasm-part-01/

[4] R. Vaño, I. Lacalle, et al., "Cloud-native workload orchestration at the edge: A deployment review and future directions", *Sensors*, vol. 23, no. 4, p. 2215, 2023. DOI: 10.3390/s23042215

[5] M. Sebrechts, J. Van der Auwera, B. Volckaert, and F. De Turck, "Adapting Kubernetes Controllers to the Edge: On-Demand Control Planes Using Wasm and WASI", in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2022, pp. 1–9.

[6] T. Goethals, F. De Turck, and B. Volckaert, "FLEDGE: Kubernetes compatible container orchestration on low-resource edge devices", in *Internet of Vehicles. Technologies and Services Toward Smart Cities*, ser. Lecture Notes in Computer Science, vol. 11894, Cham: Springer, 2019, pp. 215–230. DOI: 10.1007/978-3-030-38651-1_16

[7] KubeEdge Project, "KubeEdge: Kubernetes Native Edge Computing Framework", 2024, Accessed: Feb. 20, 2026. [Online]. Available: https://kubeedge.io

[8] E. Tinto, "Time-Predictable Runtime Infrastructure for the Edge-Cloud Continuum: Live Migration of Software Components Across Heterogeneous Nodes", M.S. thesis, University of Padova, 2024.

[9] wasmCloud Project, "wasmCloud Documentation", 2024, Accessed: Feb. 20, 2026. [Online]. Available: https://wasmcloud.com/docs/

[10] M. Fisher and M. Butcher, "Introducing Krustlet, the WebAssembly Kubelet", 2020, Accessed: Feb. 20, 2026. [Online]. Available: https://deislabs.io/posts/introducing-krustlet/

[11] LF Edge Project, "Ocre: Open Container Runtime for the Edge", 2023, Accessed: Feb. 20, 2026. [Online]. Available: https://lfedge.org/projects/ocre/

[12] Rancher Labs / SUSE, "K3s: Lightweight Kubernetes", 2024, Accessed: Feb. 20, 2026. [Online]. Available: https://docs.k3s.io/

[13] S. Wallentowitz, B. Kersting, and D. M. Dumitriu, "Benchmarking WebAssembly for Embedded Systems", *ACM Transactions on Architecture and Code Optimization*, pp. 1–21, 2025. DOI: 10.1145/3736169

[14] M. Has, T. Xiong, F. Ben Abdesslem, and M. Kusek, "WebAssembly on Resource-Constrained IoT Devices: Performance, Efficiency, and Portability", 2025. arXiv: 2512.00035 [cs.PF].

[15] Siemens AG, "SIMATIC IPC427E Industrial Edge Device", Accessed: 2026-02-20, Siemens, 2024. [Online]. Available: https://www.dex.siemens.com/edge/manufacturing-process-industries/simatic-ipc427e-industrial-edge-device

[16] Rancher by SUSE, "Rancher manager documentation", Accessed: 2026-02-10, 2024. [Online]. Available: https://ranchermanager.docs.rancher.com/

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org