Enforcing Service Stability for WebAssembly Extended Reality Workloads at the Edge: A Quality of Service Aware Orchestration Framework

Gustavo Spadotto Jardim[1], Bruno Oliveira Silvestre[2], Kleber Vieira Cardoso[2], Matheus Pires[2], Sand Luz Corrêa[2], Fábio Verdi[3], Cristiano Bonato Both[1]

[1]Universidade do Vale do Rio dos Sinos (UNISINOS), São Leopoldo – Brasil
[2] Universidade Federal de Goiás (UFG), Goiânia – Brasil
[3] Universidade Federal de São Carlos (UFSCar), São Carlos – Brasil

{brunoos, kleber, sandluz}@ufg.br, matheuspires23@discente.ufg.br,
verdi@ufscar.br, gustavosj@edu.unisinos.br, cbboth@unisinos.br

Abstract. *Modern Extended Reality (XR) applications require stringent low-latency processing that often exceeds the capabilities of end-user devices, necessitating computation offloading to the network edge, which is addressed by Multi-access Edge Computing (MEC). However, the resource-constrained and heterogeneous nature of MEC poses significant orchestration challenges. In this context, traditional distributed application orchestration and load balancing methods can lead to severe Quality of Service (QoS) degradation for immersive applications. This work addresses this gap by proposing a QoS-aware orchestration framework designed specifically for distributed WebAssembly (Wasm) computer vision workloads. We first provide a comparative analysis of Wasm runtimes against Docker containers, demonstrating that Wasm significantly reduces the memory footprint while maintaining comparable inference performance. Taking advantage of these benefits, we evaluate a custom orchestration mechanism that uses active QoS probing to enforce strict performance thresholds. The results reveal that the proposed orchestrator successfully enforces service stability. This evaluation shows that combining lightweight runtimes with active QoS-aware orchestration is a prerequisite for viable XR applications at the edge.*

1. Introduction

Extended Reality (XR) applications, encompassing Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR), have emerged as a transformative force in industrial and academic research [Rauschnabel et al. 2022, Yu et al. 2023]. These applications exhibit exceptional computational demands for real-time computer vision tasks, such as object detection [Alriksson et al. 2021]. Despite advancements in mobile hardware, end-user devices remain heavily constrained by battery life, thermal dissipation, and processing power, often rendering them incapable of sustaining high-fidelity XR experiences for prolonged periods [Satyanarayanan 2017].

A standard approach to mitigate these hardware limitations is to offload resource-intensive processes to cloud servers [Satyanarayanan 2017]. However, the average Round-Trip Time (RTT) to centralized public cloud instances often exceeds 74 ms [Li et al. 2010].

This latency is unacceptable for immersive applications, which typically require motion-to-photon delays below 16 ms to ensure stability and prevent user discomfort, e.g., motion sickness [Satyanarayanan 2017]. This fundamental physical constraint requires the adoption of Multi-access Edge Computing (MEC), which positions computational resources within the Radio Access Network (RAN) to minimize transmission latency in relation to the Mobile Cloud Computing (MCC) [Ali et al. 2018, Hu et al. 2015, Pham et al. 2020].

Although MEC effectively reduces network propagation delay, it introduces new challenges related to distributed resource management, reliability, and the coexistence of heterogeneous architectures [Pham et al. 2020]. Unlike MCC, which offers virtually elastic resources, edge nodes are heterogeneous, geographically dispersed, and resource-constrained [Mach and Becvar 2017]. In this environment, the application runtime's efficiency becomes critical. Traditional containerization solutions, such as Docker, often incur significant memory overhead and cold-start latencies. As a high-performance alternative, WebAssembly (Wasm) offers a portable binary format with minimal resource footprint, making it a potential catalyst for efficient edge computing [Haas et al. 2017, Hoque and Harras 2022]. However, runtime efficiency alone is insufficient to guarantee Quality of Service (QoS). Standard orchestration platforms such as Kubernetes (K8s) rely on static resource requests (e.g., Central Processing Unit (CPU) and Random Access Memory (RAM) availability) to make scheduling decisions. As demonstrated in this work, this "resource-blind" approach often leads to cluster saturation when handling continuous streams of Computer Vision (CV) data, resulting in severe performance degradation and data loss. To maintain the strict latency requirements of XR, orchestration must evolve from static, resource-based to active, QoS-aware decision-making.

To address these orchestration challenges, it is necessary to build upon recent advancements in the field. The existing literature has established the foundational performance benefits of Wasm in edge environments [Chennubhotla et al. 2025] and has proposed architecture-aware scheduling to address hardware compatibility [Fan et al. 2025]. Other investigations emphasized decentralized management for 6G [Blanco et al. 2024] or focused on network-layer QoS optimization [Liu and Herranz 2023]. However, a critical gap remains: these works do not provide an orchestration framework for active QoS monitoring specifically tailored to XR applications at the edge. Moreover, the authors often target static hardware attributes or network metrics, leaving the compute runtime's dynamic performance unaddressed. Motivated by this gap, this research first evaluates Wasm to confirm its performance and portability advantages for CV tasks.

Leveraging these benefits, we proceed to develop a custom orchestration framework. Our solution is validated through a distributed cluster using Lightweight Kubernetes (K3s) and submitted to a simulated orchestration scenario to rigorously assess its ability to maintain stability under the limited resources typical of the edge. Our work makes two primary contributions. First, it investigates the performance viability of Wasm as a runtime for edge-based CV workloads, demonstrating that it reduces memory footprint compared to Docker while maintaining comparable performance. The proposal and implementation of a custom orchestration framework, composed of a main orchestrator logic at the parent node, backed by latency probes and QoS-monitoring applications deployed at the worker nodes. Experimental results show that while standard K3s scheduling degrades server performance, the proposed orchestrator ensures service stability to maintain QoS.

The remainder of this article is organized as follows. Section 2 establishes the foundational concepts of this work. Section 3 reviews related work and identifies gaps in current literature. Section 4 details the design and implementation of the proposed QoS-aware orchestration framework. Section 5 presents the experimental methodology and a comparative analysis of Wasm versus Docker, followed by the stability evaluation of the custom orchestrator. Finally, Section 6 concludes the paper and outlines future research directions.

2. Background

This section establishes the foundational concepts and technologies of this research. It begins by defining the computational demands of CV within the context of XR. It explores the evolution of Distributed Computing through MEC as a solution to latency constraints. Subsequently, it examines Wasm as a lightweight execution environment for computational offloading, highlighting its advantages over traditional methods. Finally, the section details the container orchestration landscape, focusing on Docker and K8s, while identifying the scheduling limitations in standard implementations.

2.1. Computer Vision and Object Detection

XR encompasses a range of digital reality experiences, including AR, VR, and MR [Rauschnabel et al. 2022]. These concepts describe approaches that enhance user experiences with applications, devices, and their interactions with the real world [Brigham 2017]. Devices, such as the Apple Vision Pro and Meta Quest, allow users to engage with technology by rendering elements as if they were physically present. However, the processing and rendering tasks performed by these devices are highly resource-intensive, leading to limited usage times or compromising comfort by requiring larger batteries.

A core capability of XR equipment is object detection, the CV process of identifying and locating objects within an environment [Cortes et al. 2024]. This process enables XR applications to understand spatial contexts, allowing virtual elements to interact naturally with the physical world—for example, placing virtual furniture on recognized surfaces. Moreover, this object identification typically relies on deep learning models, such as Convolutional Neural Networks (CNNs), which are computationally intensive.

Object detection requires substantial processing power to perform real-time analysis. Offloading these tasks to the cloud is often unviable due to latency constraints, and VR applications usually require motion-to-photon latencies below 16 ms to prevent user discomfort [Satyanarayanan 2017]. This threshold is virtually impossible for distant cloud data centers to meet, where round-trip times often exceed 50–100 ms, thereby justifying the need for computation at the network edge, which is a subject further developed by MEC and is explored in Subsection 2.2

2.2. Distributed Computing and MEC

Distributed systems consist of multiple computing units connected through a network that collaboratively process applications by executing tasks across nodes [Jiang 2016]. In the context of XR, this paradigm shifts towards MEC, which extends cloud computing capabilities to the edge of the network, specifically within the RAN in close proximity to end-users [Abbas et al. 2018]. MEC addresses the latency challenges inherent in distributed systems by processing data near the User Equipment (UE), improving service delivery for

high-bandwidth, low-latency applications [Hu et al. 2015]. In distributed computing architectures, distributing tasks to minimize latency and balance load is a critical challenge. Unlike MCC, edge environments are heterogeneous and resource-constrained, requiring efficient orchestration mechanisms to manage the variability in user mobility and workload intensity. Figure 1 describes an example of how a distributed edge architecture may look, with multiple UEs connected to spread edge nodes geographically dispersed.

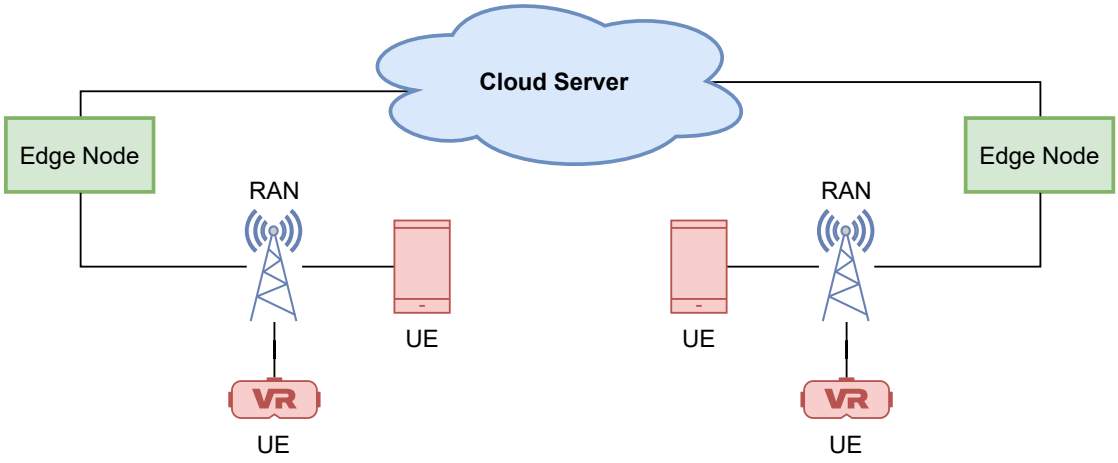

Figure 1. Distributed edge architecture example.

2.3. Docker and K8s

Containerization has become the standard for deploying distributed applications. Docker packages applications and their dependencies into standardized units, ensuring consistency across environments. However, traditional Docker containers can introduce significant cold-start latency and memory overhead, which impacts real-time edge applications. To manage these containers at scale, K8s provides an orchestration platform for automating deployment, scaling, and management [Burns et al. 2016]. The K8s architecture consists of a control plane (master node), which makes global scheduling decisions, and worker nodes, which run the `kubelet` and container runtime to execute workloads [Kubernetes 2024]. Additionally, K8s is enabled to run Wasm containers through extension of its Container Runtimes [Kubernetes 2024].

For MEC, full K8s distributions are often heavy. In this case, K3s is a lightweight, Cloud Native Computing Foundation-certified K8s distribution designed specifically for resource-constrained environments [Fan et al. 2025]. It reduces the memory footprint by packaging components into a single binary and removing non-essential cloud providers. Despite these optimizations, the default K8s scheduler has limitations for XR workloads at the edge, such as relying on resources availability (e.g., CPU and RAM) rather than real-time performance metrics or dynamic network conditions [Kubernetes 2026]. In heterogeneous edge scenarios, simply fitting a workload onto a worker node is insufficient; the orchestrator must ensure the node meets specific QoS requirements. This limitation necessitates the development of custom scheduling logic that is not just resource-aware, but that instead relies on different approaches that can meet strict QoS requirements.

2.4. WebAssembly and Offloading

Wasm is a low-level binary format designed to be independent of hardware and platform, allowing code compiled from languages, such as C++, Rust, or Python to execute seamlessly across heterogeneous environments [Haas et al. 2017]. While Wasm was originally developed for the Web, its system interface, WebAssembly System Interface (WASI), extends its capabilities to non-Web environments by providing standardized access to system resources, such as files and networking [Alliance 2024]. Crucially for CV workloads, extensions such as WASI Neural Network (WASI-NN) enable Wasm modules to perform high-performance inference using hardware acceleration, bridging the gap between portability and the computational demands of CNNs.Wasm is particularly well-suited for offloading, that in this context could be defined as the delegation of compute-intensive task from a client mobile device to a remote edge node. [Chennubhotla et al. 2025] highlights the efficiency gains of this approach in edge scenarios, with smaller memory footprint and image size and faster execution speeds. These characteristics are appropriate for deployments on resource-constrained edge nodes, minimizing the overhead typically associated with containerization. In light of these advantages, Wasm stands out as a great technology that XR applications may leverage to optimize workload offloading to MEC.

3. Related Work

Recent advancements in edge computing and container orchestration have driven the exploration of more efficient runtime environments and sophisticated management strategies. To establish the state of the art for this research, a targeted literature review was conducted focusing on two primary domains: the orchestration of XR workloads at the network edge and the comparative performance benchmarking of Wasm runtimes against traditional Docker containers. The articles selected for this review represent key contributions in runtime efficiency, orchestration approaches, and QoS-aware network integration.

As MEC demands increase, the limitations of traditional containerization in resource-constrained environments have become apparent. [Chennubhotla et al. 2025] evaluate Wasm as a viable alternative to Docker for serverless functions on the edge. Their benchmarking across heterogeneous devices demonstrates that Wasm-based functions offer significant performance advantages over Docker, achieving 31.4% faster average execution speeds. Additionally, the study reports a 59.5% reduction in memory footprint and image sizes that are 86.5% smaller than their Docker counterparts. These findings highlight Wasm's efficacy in resource-scarce environments.

To leverage the benefits of Wasm while maintaining compatibility with existing container ecosystems, hybrid orchestration frameworks have emerged. [Fan et al. 2025] propose a lightweight orchestration framework based on K3s that seamlessly manages both Docker containers and Wasm modules. By integrating the WebAssembly Micro Runtime (WAMR) via a custom containerd-shim (an intermediary adapter that abstracts the runtime execution lifecycle, allowing the container manager to control diverse environments such as Wasm without direct dependency) and employing an architecture-aware scheduling strategy, their solution optimizes workload placement based on hardware suitability, such as distinguishing between ARM and x86 architectures. This approach optimizes resource allocation, reducing memory and CPU overhead while improving startup latency and storage efficiency.

Effective orchestration at the MEC also requires tight integration with network

QoS mechanisms. Current research, such as the one led by [Liu and Herranz 2023], addresses QoS by optimizing the network layer, ensuring that priority traffic receives dedicated bandwidth and specific transport characteristics. While this is essential for reducing transmission latency, end-to-end performance cannot be guaranteed if the compute runtime itself introduces processing bottlenecks. This work complements network-centric QoS approaches by targeting the application layer and utilizes a custom scheduler to ensure workloads are placed on nodes capable of meeting processing deadlines, leveraging the efficient Wasm runtime to address the computational component of the QoS equation.

The complexity of managing massive network slices and distributed resources requires decentralized architectural approaches. [Blanco et al. 2024] present the MonB5G framework, which distributes Artificial Intelligence (AI)/Machine Learning (ML)-driven management components, such as Monitoring System (MS), Analytics Engine (AE), Decision Engine (DE), and Actuator (ACT),across the RAN, Edge, Cloud, and Core domains. Their work demonstrates that distributed management utilizing Federated Learning (FL) significantly reduces monitoring overhead and improves energy efficiency compared to centralized algorithms. This work reinforces the necessity for distributed computing solutions in orchestrating complex, QoS-demanding workloads across next-generation networks.

While literature establishes the foundational performance benefits of Wasm in edge environments, our work advances the field by operationalizing these metrics through dynamic, application-specific orchestration. Unlike [Chennubhotla et al. 2025], who primarily benchmark Wasm's efficiency against Docker to establish static baselines, our research integrates those advantages into a custom K8s scheduler capable of real-time decision-making. Moreover, as [Fan et al. 2025] propose architecture-aware scheduling to address hardware compatibility, our work introduces QoS-aware orchestration that prioritizes dynamic performance metrics over static hardware attributes. Rather than simply matching workloads to specific architectures, our proposed scheduler actively evaluates real-time constraints, such as inference latency threshold, specifically tailored to the stringent requirements of CV workloads. Additionally, [Blanco et al. 2024] emphasizes decentralized architectures for 6G to reduce overhead, and our work focuses on the orchestrator's logic to ensure that distributed resources meet specific application Service Level Agreements (SLAs). Finally, whereas [Liu and Herranz 2023] focuses on bridging orchestration with network-layer QoS, this approach targets the compute runtime itself, leveraging Wasm's benefits to ensure that application processing aligns dynamically with end-to-end performance goals.

4. QoS-aware Orchestration Framework for Computer Vision Applications

To optimize the orchestration of CV workloads in edge environments, this work moves beyond traditional resource-based scheduling toward active QoS-aware placement. Operating as a centralized control plane, the designed orchestrator mediates between client requests and distributed heterogeneous nodes. The orchestrator is based on a three-stage mechanism, consisting of (i) filtering, (ii) evaluating, and (iii) deploying, designed to account for network proximity and computational capability. Figure 2 illustrates the architectural framework and the interaction between the centralized control plane and the distributed edge entities. By decoupling network-level constraints from application QoS requirements, this mechanism ensures that low-latency CV workloads are mapped only to nodes capable of sustaining the necessary service performance for real-time execution.

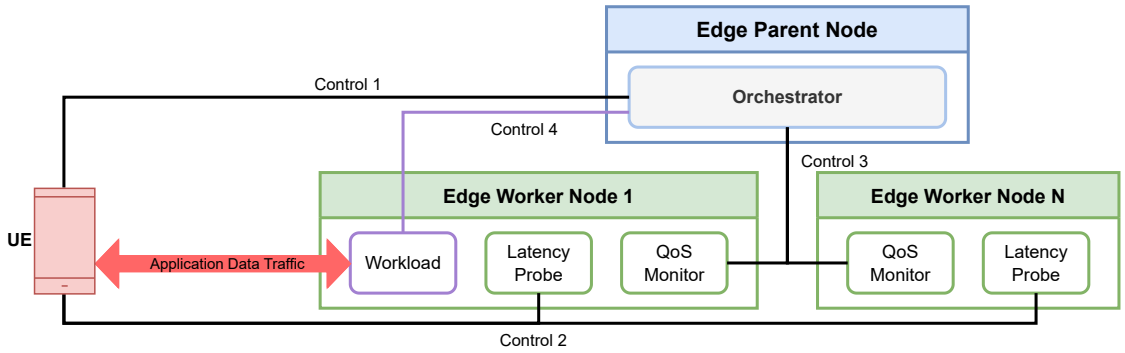

Figure 2. QoS-aware orchestrator framework base design.

The interaction between the components shown in the framework design is ruled by a set of control interfaces, as detailed in Table 1. The system divides responsibilities between the UE and the orchestrator to ensure that only nodes that meet the requirements are considered. This approach allows the filtering phase to reduce overhead by removing distant nodes before intensive performance evaluations begin.

Table 1. Description of Control Interfaces in the Orchestration Framework.

Interface	Actors	Function	Stage
Control 1	UE ↔ Orchestrator	(a) Negotiation of deployment requests; (b) Retrieval of available node lists; (c) Reception of client-side filtered nodes; (d) Node allocation decision; (e) Request rejection if QoS is unmet	Filtering
Control 2	UE ↔ Latency Probe	(a) Network latency measurement (RTT); (b) Client-side filtering of candidates based on latency	Filtering
Control 3	Orchestrator ↔ QoS Monitor	(a) Active QoS monitoring; (b) Node capacity verification; (c) Enforcement of performance thresholds	Evaluation
Control 4	Orchestrator → Worker	(a) Wasm workloads instantiation; (b) Lifecycle management of workloads	Deployment

The deployment process is divided into a client-side network for effective placement filtering and a server-side inference performance evaluation. The process begins when a UE requests deployment. The orchestrator first returns a list of all available nodes in the cluster. The UE performs a network latency test against these nodes to determine network proximity. Nodes that exceed a pre-defined network latency threshold (θ_{net}) are discarded by the client to prevent deploying to distant nodes that cannot support real-time interaction, regardless of their compute power.

Subsequently, the UE sends the filtered list of candidate nodes back to the orchestrator for the final selection. The orchestrator validates the real-time computational

health of these candidates through active performance probes. A node n_i is considered eligible only if it satisfies both a local capacity constraint ($CurrentWorkloads(n_i) < MaxCapacity(n_i)$) and a dynamic performance threshold, $L_{inf}(n_i) \leq \theta_{inf}$, where L_{inf} is the measured inference latency. Consequently, the best-fit node n^* is identified by minimizing the inference delay among eligible candidates, as shown in the equation:

$$n^* = \arg \min_{n_i \in Eligible} \{L_{inf}(n_i)\} \tag{1}$$

This threshold-based logic enforces a strict admission control policy. If no node in the filtered list satisfies the inference latency threshold θ_{inf}, the deployment request is rejected. This fail-fast mechanism prevents the instantiation of workloads that would violate QoS requirements, protecting the cluster from degradation and ensuring stable service for admitted clients. To accurately quantify L_{inf}, an active probing mechanism transmits fragmented sample data (e.g., a 640×640 image) to a monitoring service on the target node, providing a high-fidelity indicator of current inference capability. The system manages a dedicated network port pool per node to further minimize communication overhead. The complete decision-making logic is outlined in Algorithm 1.

Algorithm 1 SCHEDULEWORKLOAD(candidates, θ_{inf})

Require: candidates: list of network-filtered nodes
Require: θ_{inf}: maximum inference latency threshold (ms)
Ensure: targetNode: selected node or null

$targetNode \leftarrow$ `null`
$minLatency \leftarrow \infty$
for all node n in candidates **do**
    if $ActiveWorkloads(n) < CapacityLimit(n)$ **then**
        $L_{inf} \leftarrow$ `measure_inference_latency`$(n.ip)$
        if $L_{inf} \leq \theta_{inf}$ **and** $L_{inf} < minLatency$ **then**
            $minLatency \leftarrow L_{inf}$
            $targetNode \leftarrow n$
        end if
    end if
end for
if $targetNode \neq$ `null` **then**
    $commPort \leftarrow$ `allocate_network_port`$(targetNode)$
    Instantiate Workload($targetNode$, $commPort$)
else
    Reject Deployment Request ▷ QoS cannot be guaranteed
end if
return $targetNode$

5. Experiments and Results

A prototype was developed to validate the proposed orchestration framework and the Wasm execution model, comprising an object detection application written in Rust and compiled for the *wasm32-wasip2* target. The runtime environment was established using *Wasmtime* and *wasmtime-wasi-nn* version 42.0.0. To enable machine learning in-

ference within the orchestration layer, a modified version of *runwasi*[1] was implemented to support *wasi-nn*. Additionally, the project leverages PyTorch 2.7.1 CPU version as the underlying ML inference backend. This workload was deployed on a heterogeneous physical testbed orchestrated by K3s, selected for its suitability in managing resource-constrained environments. Although the default K8s scheduler is not inherently designed for the resource-intensive demands of edge-based applications, this work evaluates its performance to establish a necessary baseline for the standard orchestration stack. The infrastructure consists of three primary nodes: a host that serves as the control plane and the most powerful edge worker, a Virtual Machine (VM), and a laptop that functions as a resource-restricted edge worker. These machine specifications are detailed in Table 2.

Table 2. Hardware Specifications of the Distributed Testbed.

Node Name	Role	CPU Model	RAM	Environment
Master	Control Plane	AMD 5500X3D	12 GB	WSL2 (Kernel 6.6)
Worker 1	Physical Worker	Intel i5-1135G7	16 GB	Ubuntu 24.04
Worker 2	Virtual Worker	AMD 5500X3D	4 GB	Xubuntu 24.04

Using this infrastructure, this study conducts three distinct experiments to evaluate the technical viability and orchestration efficiency of Wasm in edge environments. First, a *YOLOv8-based* object detection task is utilized to evaluate computational overhead and memory footprint. The task processes a 30-second video at 1280×720 resolution, rendering bounding boxes, using the application deployed across three environments: a standalone Wasm application running directly on a host, a Docker container, and a Wasm container. Each configuration executes the application 30 times. Second, deployment agility is quantified by measuring the ”cold start” latency, comparing the time required for Docker and Wasm container pods to transition from a deployment request to an active processing state. These experiments were repeated 30 times.

Finally, the orchestrator evaluation was conducted by subjecting the cluster to a stress test comparing a custom latency-aware scheduler with the default K8s scheduler under an incremental load pattern. This simulation generated requests for 10 UEs starting with an initial burst of three at $T + 0s$, followed by a gradual load of seven additional UEs between $T + 5s$ and $T + 40s$ to evaluate admission control and performance-based routing capabilities. We repeated the experiment 10 times to guarantee the reliability of the result. Moreover, our prototype is publicly available[2], enabling the reproducibility of the experiments and results.

5.1. Wasm vs Docker Container vs Wasm Container

Containerized Wasm emerged as the most efficient runtime, delivering the highest average frame rate and the lowest remote inference latency[3]. Docker trailed by a negligible margin, while the standalone Wasm runtime exhibited a 5–6% performance deficit. Despite these variances in inference tasks, system overheads (decoding, encoding, rendering)

[1]The project that generates containerd shims for *Wasmtime* and other Wasm runtimes.

[2]https://github.com/gustavojardim/wasm-object-detection

[3]The total duration required for the Wasm application to perform inference on an image.

remained identical across all platforms. Figures 3 and 4 substantiate these findings, illustrating the consistent lead of the containerized Wasm approach across both frame rate and remote inference latency metrics.

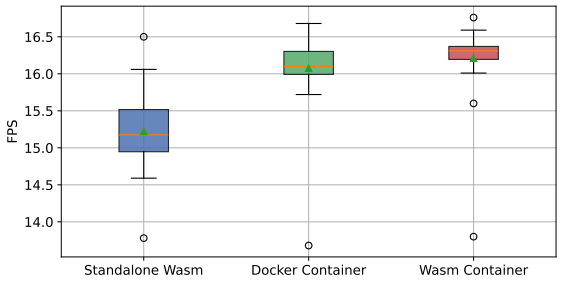

Figure 3. Wasm vs Docker vs Wasm Container FPS.

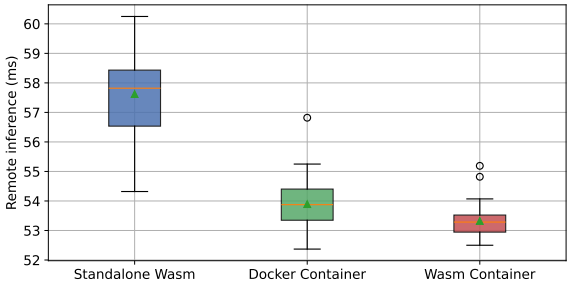

Figure 4. Wasm vs Docker vs Wasm Container Inference Latency.

Table 3 summarizes the quantitative analysis of resource usage, such as memory footprints, storage requirements, CPU usage, and cold-start latency (not applicable to Standalone Wasm) across the deployment methods. The data highlights distinct performance tiers, demonstrating the significant reduction in overhead achieved by the Wasm-based approaches compared to the traditional Docker environment for RAM and storage usage. CPU usage remained uniform across all instances. This consistency confirms that neither the containerization layer nor the Wasm runtime introduces significant processing overhead for this specific workload. Cold-start latency was also very similar between Wasm and Docker containers. Storage discrepancies are due to Wasm containers requiring only the model files, as the libraries they need (e.g., PyTorch, Rust, Wasmtime) are consumed from the host, whereas Docker requires the models plus the cited libraries.

Table 3. Resource Efficiency Comparison across Deployment Methods

Deployment Method	Baseline RAM	Storage Size	Avg. CPU Usage	Cold-Start Latency
Standalone Wasm	≈ 242 MB	≈ 2 MB	≈ 37.62%	Not applicable
Wasm Container	≈ 295 MB	≈ 27 MB	≈ 37.08%	≈ 2.69 s
Docker	≈ 475 MB	≈ 4.65 GB	≈ 37.80%	≈ 2.62 s

5.2. Custom Orchestration Results

The results demonstrated that Wasm runtimes offer a significant advantage over traditional containerization, reducing memory and storage overhead while maintaining near-native inference performance. More critically, the results highlight a fundamental gap in standard orchestration: the default K8s scheduler's approach led to a high average number of timed-out frames under load, whereas our custom performance-degradation-aware orchestrator virtually eliminated these service disruptions. By prioritizing real-time application metrics, the custom orchestrator effectively maintained the cluster's functional capacity while ensuring stability and low jitter, both essential for high-fidelity edge vision applications.

The most significant findings emerged from the stress test. The K8s default scheduler followed an "allow everything" policy, leading to performance degradation. The cus-

tom orchestrator, utilizing inference-latency-aware admission control, successfully identified cluster saturation. Although it rejects requests to protect cluster health, the admitted clients maintained better performance stability than the K8s default scheduler. Figure 5 shows that, although K8s accepts all the requests that it receives, it deployed fewer workloads than the custom orchestrator due to performance degradation. Regarding deployment overhead, Figure 6 highlights the discrepancy between the mean and median metrics due to atypical values. While the mean deployment time was approximately 9.25 seconds, this value was heavily influenced by sporadic outliers, with singular events exceeding 100 seconds. These extreme values are likely a symptom of missing timeout logic within the custom orchestrator. However, this behavior is comprehensible given the current experimental state of the prototype. To better reflect nominal system performance, the median deployment time provides a more accurate representation, stabilizing at approximately 5.37 seconds.

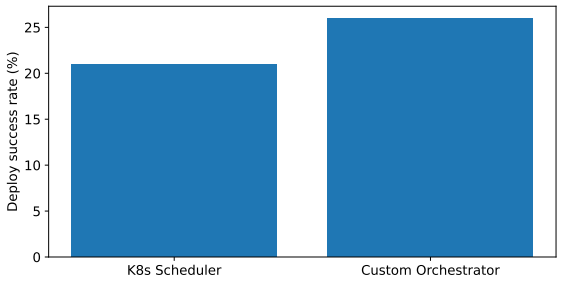

Figure 5. K8s Scheduler vs. Orchestrator deploy success rate.

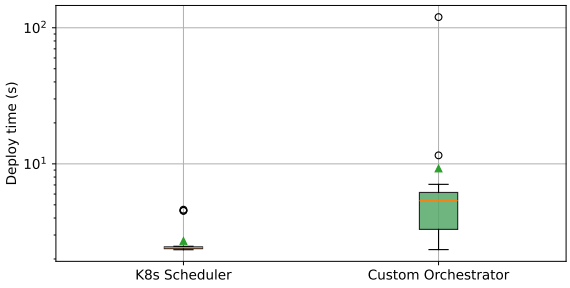

Figure 6. K8s Scheduler vs. Orchestrator deploy time.

The analysis of inference latency and frame rate, as shown in Figures 7 and 8, provides empirical validation of the custom orchestrator's superior stability, demonstrating lower variance than the default K8s scheduler. In contrast to the default scheduler, which exhibits FPS fluctuations between 7 and 17 and latency ranging from 52 ms to 102 ms, the custom orchestrator exhibits highly deterministic performance behavior.

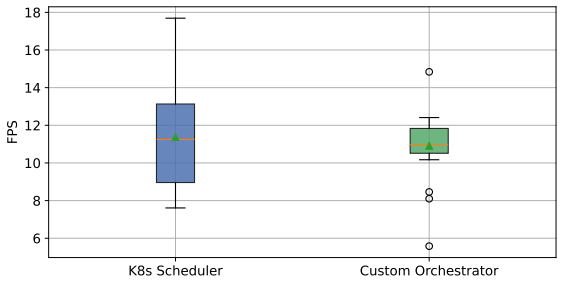

Figure 7. K8s Scheduler vs. Orchestrator FPS.

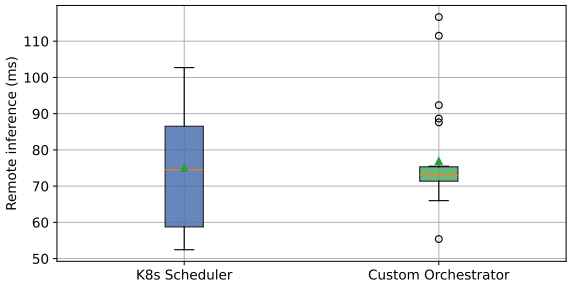

Figure 8. K8s Scheduler vs. Orchestrator Inference Latency

By keeping FPS and remote inference latency within a tight range, the proposed solution effectively preserves service stability. Nonetheless, it is essential to note that the orchestration process introduced additional overhead to the cluster, and each worker node was required to run two dedicated pods (one for latency probing and another for QoS

monitoring) while the master node, which also functioned as a worker, hosted the core orchestration pod. This experiment demonstrates a strategic trade-off, exchanging a negligible margin of theoretical peak performance for a consistent QoS stability, guaranteeing a reliable video stream for all admitted clients.

Because FPS and latency metrics exclude timed-out frames, the reported averages do not reflect the qualitative user experience degradation. Qualitative observations during tests with the K8s default scheduler revealed severe performance issues, characterized by frequent stuttering and complete video freezes. This instability is quantitatively demonstrated in Figure 9, which reports the number of timed-out frames per run. In this experimental setup, a 'run' consists of a single client processing 188 video frames for object detection. Figure 9 quantifies this instability. In a standard 188-frame run, the K8s default scheduler averaged 80.8 timed-out frames (Standard Deviation (SD)=93.1), with some runs failing completely. In contrast, the custom orchestrator demonstrated exceptional stability, averaging only 0.08 missed frames (SD=0.27).

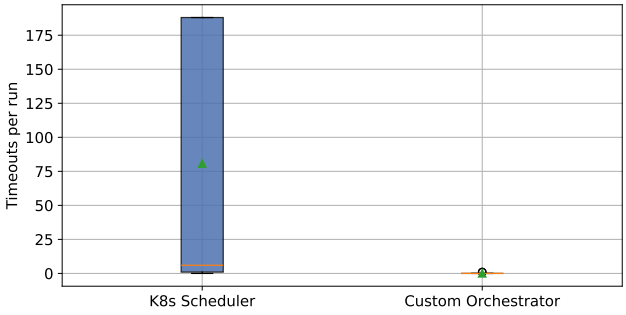

Figure 9. Timeouts per run.

6. Conclusions

In this work, we presented a QoS-aware orchestration framework designed for distributed Wasm CV applications at the edge. By leveraging the lightweight nature of Wasm and implementing the framework on a distributed computing cluster that simulated edge nodes to execute XR workloads, we verified stable QoS metrics in a replicated resource-constrained environment. Our comparative analysis indicates that Wasm offers significant efficiency gains over traditional containerization for edge computing. While maintaining performance parity with Docker for "cold start" times and inference latency, the Wasm runtime operated with a substantially lower memory footprint and used smaller deployment artifacts, optimizing the limited storage and bandwidth available at the edge. The experimental results regarding orchestration highlighted the limitations of applying standard, resource-blind scheduling to real-time video applications. The default K8s scheduler, operating without application-layer performance feedback, caused cluster saturation, degraded service performance, and a significant number of timed-out frames. In contrast, the proposed custom orchestrator, by enforcing strict application performance constraints, successfully acted as a performance gatekeeper. Although this approach rejected excess workload requests to preserve cluster health, it effectively maintained data integrity for admitted clients, achieving high stability with negligible frame loss.

These findings support the QoS-aware orchestration framework as a valid approach for XR applications and potentially other edge-running computer vision work-

loads, prioritizing execution stability and predictable performance over raw resource availability. As future work, we plan to conduct experiments at larger scales and under more diverse and intensive workloads to evaluate further the framework's scalability, robustness, and effectiveness. Additionally, we intend to validate the proposed approach in real-world scenarios, such as a 5G network with connected mobile devices, to assess its performance and practicality under realistic operational conditions.

7. Acknowledgments

This work was supported by Ericsson Telecomunicações Ltda., and by the São Paulo Research Foundation (FAPESP), grant 2021/00199-8, CPE SMARTNESS, and project PORVIR-5G (Grants No. 2020/05182-3). Furthermore, the work was also partially supported by CNPq Grants Nos. 308720/2025-3.

References

[Abbas et al. 2018] Abbas, N., Zhang, Y., Taherkordi, A., and Skeie, T. (2018). Mobile Edge Computing: A Survey. *IEEE Internet of Things Journal*, 5(1):450–465.

[Ali et al. 2018] Ali, S. S. D., Ping Zhao, H., and Kim, H. (2018). Mobile Edge Computing: A Promising Paradigm for Future Communication Systems. In *TENCON 2018 - 2018 IEEE Region 10 Conference*, pages 1183–1187.

[Alliance 2024] Alliance, B. (2024). WebAssembly System Interface (WASI). Accessed on July 12, 2025.

[Alriksson et al. 2021] Alriksson, F., Kang, D. H., Phillips, C., Pradas, J. L., and Zaidi, A. (2021). XR and 5G: Extended reality at scale with time-critical communication. *Ericsson Technology Review*, 2021(8):2–13.

[Blanco et al. 2024] Blanco, L., Zeydan, E., Barrachina-Muñoz, S., Rezazadeh, F., Vettori, L., and Mangues-Bafalluy, J. (2024). A Novel Approach for Scalable and Sustainable 6G Networks. *IEEE Open Journal of the Communications Society*, 5:1673–1692.

[Brigham 2017] Brigham, T. J. (2017). Reality Check: Basics of Augmented, Virtual, and Mixed Reality. *Medical Reference Services Quarterly*, 36(2):171–178. PMID: 28453428.

[Burns et al. 2016] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM*, 59(5):50–57.

[Chennubhotla et al. 2025] Chennubhotla, A., Tewari, S., Somvanshi, H., M.J, C., and C, D. (2025). Moving Past Docker for Serverless: WebAssembly for Kubernetes Clusters at the Edge. In *2025 10th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, pages 560–565.

[Cortes et al. 2024] Cortes, D., Bermejo, B., and Juiz, C. (2024). The use of CNNs in VR/AR/MR/XR: a systematic literature review. *Virtual Reality*, 28(3):154.

[Fan et al. 2025] Fan, X., Li, J., and Deng, C. (2025). Hybrid WebAssembly-Container Orchestration in Embedded Heterogeneous Scenarios. In *2025 5th International Conference on Artificial Intelligence and Industrial Technology Applications (AIITA)*, pages 1525–1530.

[Haas et al. 2017] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2017). Bringing the web up to speed with WebAssembly. *SIGPLAN Not.*, 52(6):185–200.

[Hoque and Harras 2022] Hoque, M. N. and Harras, K. A. (2022). WebAssembly for Edge Computing: Potential and Challenges. *IEEE Communications Standards Magazine*, 6(4):68–73.

[Hu et al. 2015] Hu, Y. C., Patel, M., Sabella, D., Sprecher, N., and Young, V. (2015). Mobile Edge Computing A key technology towards 5G. Technical report, ETSI: European Telecommunications Standards Institute.

[Jiang 2016] Jiang, Y. (2016). A Survey of Task Allocation and Load Balancing in Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):585–599.

[Kubernetes 2024] Kubernetes (2024). Kubernetes Architecture. Accessed: 2025-07-12.

[Kubernetes 2026] Kubernetes (2026). Kubernetes documentation. `https://kubernetes.io/docs`. Acessed in: February 6th, 2026.

[Li et al. 2010] Li, A., Yang, X., Kandula, S., and Zhang, M. (2010). CloudCmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 1–14, New York, NY, USA. Association for Computing Machinery.

[Liu and Herranz 2023] Liu, Y. and Herranz, A. H. (2023). Enabling 5G QoS configuration capabilities for IoT applications on container orchestration platform. In *2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 63–68.

[Mach and Becvar 2017] Mach, P. and Becvar, Z. (2017). Mobile Edge Computing: A Survey on Architecture and Computation Offloading. *IEEE Communications Surveys I& Tutorials*, 19(3):1628–1656.

[Pham et al. 2020] Pham, Q.-V., Fang, F., Ha, V. N., Piran, M. J., Le, M., Le, L. B., Hwang, W.-J., and Ding, Z. (2020). A Survey of Multi-Access Edge Computing in 5G and Beyond: Fundamentals, Technology Integration, and State-of-the-Art. *IEEE Access*, 8:116974–117017.

[Rauschnabel et al. 2022] Rauschnabel, P. A., Felix, R., Hinsch, C., Shahab, H., and Alt, F. (2022). What is XR? Towards a Framework for Augmented and Virtual Reality. *Computers in Human Behavior*, 133:107289.

[Satyanarayanan 2017] Satyanarayanan, M. (2017). The Emergence of Edge Computing. *Computer*, 50(1):30–39.

[Yu et al. 2023] Yu, H., Shokrnezhad, M., Taleb, T., Li, R., and Song, J. (2023). Toward 6G-Based Metaverse: Supporting Highly-Dynamic Deterministic Multi-User Extended Reality Services. *IEEE Network*, 37(4):30–38.