

# A WebAssembly-Based Framework for Decomposed Applications in the Cloud-Edge Continuum

Guilherme Mendes Vieira de Matos<sup>ufsc</sup>, Fábio Luciano Verdi<sup>ufsc</sup>

Andrew Williams<sup>ericsson</sup>

<sup>ufsc</sup> Federal University of São Carlos, Brazil

<sup>ericsson</sup> Ericsson Research, Sweden

**Abstract**—This paper presents a lightweight coordination framework for executing decomposed applications across the cloud-edge continuum using WebAssembly-based components. The framework separates application logic from execution management, allowing developers to implement only the core kernel of each stage, while the system handles deployment, routing, and data forwarding across nodes. In this context, the kernel denotes the application-specific computation, whereas the receiver component is responsible for network input processing, message handling, and traffic forwarding. Built on the WebAssembly component model, the proposed design combines reusable application components with network-aware execution support, enabling chained functions to be deployed and connected transparently across distributed nodes. We evaluate the framework using a three-stage image-processing pipeline under different placement scenarios and input sizes. The results show that the system preserves correct end-to-end execution while introducing the expected overheads of decomposition and distributed forwarding. Under controlled low-latency conditions, selective offloading is achieved with minimal distribution overhead, demonstrating the feasibility of the proposed WebAssembly component substrate for application coordination before scaling to wider heterogeneous continuum environments.

**Index Terms**—Cloud-edge computing continuum, distributed environment, application decomposition, web assembly.

## I. INTRODUCTION

The rise of edge computing and the growing availability of resources outside traditional data centers, such as base stations and local gateways, are driving the uptake of distributed systems toward the cloud-edge continuum. In this scenario, applications may span constrained devices, edge nodes, regional servers, and cloud data centers within a single execution environment, increasing both resource availability and deployment complexity across widespread locations and heterogeneous hardware [1].

In this context, previously monolithic applications are increasingly decomposed into ordered functional blocks, where the output of one block becomes the input of the next [2]. It is important to note that decomposition does not imply a traditional microservice architecture. Instead, decomposition focuses on splitting a single application into a chain of ordered functional blocks, forming a pipeline where blocks must execute in sequence and exchange data at runtime, with the goal of enabling fine-grained placement and steering across heterogeneous nodes in the continuum [3]. This becomes necessary in resource-constrained environments, where a single

node may not be able to execute the entire application within a required time window, making it essential to distribute the workload and maximize the use of available resources.”

Existing cloud-native solutions adapted to the edge present architectural limitations when executing fine-grained decomposed applications due to the heavy footprint of container-based runtimes, motivating lightweight WebAssembly-based execution substrates. Although they rely on mature orchestration stacks like Kubernetes, they typically manage resources at the pod or container level without awareness of the application’s internal block dependencies. While existing programming models attempt to reduce platform-specific adaptation, they still lack orchestration mechanisms capable of coordinating these fine-grained blocks across the continuum [1]. This limitation is amplified in the computing continuum, where developers must also account for the dynamic network layer introduced by distribution across cloud and edge nodes.

WebAssembly [4] emerges in this space as a promising building block, providing the required portability and runtime unification, but its effective use still requires orchestration mechanisms that understand both application decomposition and heterogeneous execution contexts [2]. Current research solutions leveraging Wasm have explored migration, placement, and adaptation [1], [2], [5], [6]. However, they still fall short of providing unified orchestration for both execution code and data routing. Furthermore, even when applications are written once, compilation often remains target-dependent, leaving a cross-compilation burden that developers must continue to manage manually [7].

To fill this gap we present a WebAssembly-based solution for coordinating decomposed applications across distributed nodes in the computing continuum. Our approach supports chaining, placement, and execution decisions based on runtime conditions, such as network latency and node resource availability, while enabling efficient offloading across distributed environments. Our solution uses WebAssembly [4] and the recently introduced Wasm Component Model to ensure function interface portability across programming languages [8], connecting application logic with network-aware components

In general, our architecture introduces a dedicated receiver component; by leveraging the Wasm Component Model, this receiver can be composed with any WIT-compatible function to seamlessly transform standard application kernel, i.e., the

application logic, into a network-equipped task. The receiver abstraction handles all incoming network traffic and delivers data directly to the service logic, shielding developers from low-level networking code. To manage these tasks across the continuum, an orchestrator handles high-level placement and instantiation decisions, while the system-level network components execute the actual traffic steering and data forwarding among them. Consequently, routing decisions are made automatically according to the current execution context.

Our results show that remote placement adds negligible overhead once the pipeline is decomposed. Transitioning from local to remote execution merely increases completion times from 1.996 s to 2.025 s for HD, and from 15.544 s to 15.657 s for 4K workloads, even when offloading only the heaviest stage.

The remainder of this paper is organized as follows. Section II presents the background and motivation, including industrial solutions and related work. Section III describes the framework workflow. Section IV describes the implementation and Section V reports the evaluation results. Finally, Section VI concludes the paper and outlines future work.

## II. BACKGROUND AND MOTIVATION

This section reviews related work in orchestration, portability, and deployment across the cloud-edge continuum, focusing on solutions for execution across heterogeneous architectures and runtime coordination

### A. Industrial Solution

Industrial solutions such as Kubernetes-based orchestration stacks or Open Cluster Management [9] provide mature mechanisms for managing distributed workloads in cloud environments, with strong support for deployment, scaling, and cluster-level control. However, their design is still rooted in cloud-centric assumptions, and they do not treat decomposed applications as a first-class concern. As a result, they are less suitable for the cloud-edge continuum, where latency sensitivity, resource constraints, and heterogeneous hardware make lightweight and application-aware orchestration essential. In this work, we argue that the key gap is not only deployment management, but orchestration intelligence that can operate closer to the application and the runtime.

### B. Related works

Tinto and Vardanega [5] propose a runtime infrastructure for the Computing Continuum based on Rust and WebAssembly, aiming to treat Cloud and Edge as a single resource pool and to support migration of functions while preserving their state. Their work also argues that cloud-native orchestrators are too cloud-centric to properly address the needs of the Edge, motivating the need for continuum-aware orchestration mechanisms. In this work, the focus is different: rather than supporting migration alone, we target the orchestration of decomposed applications, using WIT-based [8] composition to connect reusable components while abstracting the network

layer from developers and enabling automatic instantiation and traffic steering across distributed nodes.

Grosch et al. [1] argue that WebAssembly is a strong substrate for distributed edge-cloud systems because it is fast, portable, safe, lightweight, and suitable for modular applications with well-defined interfaces. The paper also highlights that Wasm modules can be composed into loosely coupled applications and that orchestration is still an open problem, especially because cloud-oriented frameworks do not translate cleanly to the edge and many edge applications are stateful. Although the authors argue for the use of an orchestrator, they do not implement one. In this work, the focus goes beyond using Wasm as an execution substrate: we address the coordination of decomposed applications across distributed nodes, combining metric-driven placement decisions with component model composition and a network-abstracting receiver to dynamically instantiate and connect components across the continuum.

Nurul-Haque and Harras [6] focuses on portability and migratability for edge offloading in heterogeneous femtoclouds<sup>1</sup>. The main contribution is an interpreter-based WebAssembly runtime that can live-migrate execution across operating systems and hardware architectures, addressing the need for cross-platform migration with low overhead. In this work, the goal is different: instead of migrating a single running computation, we coordinate decomposed applications as composable building blocks, handling dynamic component placement and traffic steering while abstracting network-layer complexities from the developer.

Khelifa et al. [7] explore how to make Wasm more effective for Deep Learning inference by integrating external runtimes such as TVM [10] through WASI-NN [11] and deploying the solution in Knatives<sup>2</sup>. The paper is centered on improving inference performance and portability across heterogeneous devices, especially in the presence of hardware accelerators and serverless deployment constraints. In our work, rather than optimizing a specific inference stack, we use the Wasm component model and WIT to orchestrate decomposed applications across the continuum, with automatic instantiation, composition, and traffic steering across distributed execution sites.

TABLE I: Comparison of related works and the proposed approach.

| Concepts / Works                    | [5] | [1] | [6] | [7] | This work |
|-------------------------------------|-----|-----|-----|-----|-----------|
| Computing continuum focus           | X   | X   |     | X   | X         |
| WebAssembly-based runtime           | X   | X   | X   | X   | X         |
| Migration support                   | X   |     | X   |     |           |
| Application decomposition           |     |     |     |     | X         |
| Orchestration of applications       | X   | X   |     | X   | X         |
| WIT / component model               |     |     |     |     | X         |
| Network abstraction for developers  |     |     |     |     | X         |
| Traffic steering between components |     |     |     |     | X         |

<sup>1</sup>An opportunistic mobile cloud computing framework that utilizes idle resources from proximal devices (through direct or multi-hop communication) to establish a transient local processing cluster

<sup>2</sup><https://github.com/knative/serving>

Overall, as we can see in Table I, the related work shows clear progress in making WebAssembly portable, migratable, and efficient across heterogeneous environments. However, the deployment of decomposed applications remains insufficiently addressed, especially when network handling must be transparent to developers and execution decisions must be made dynamically across the continuum. This gap motivates our approach, which combines WebAssembly component composition, runtime coordination, and traffic steering to support decomposed applications efficiently in heterogeneous cloud-edge environments.

### III. WORKFLOW

In this work, an application is represented as a composed pipeline of independent functional blocks executed in sequence. Rather than focusing on the breakdown process, which has already been extensively investigated in prior literature [3], [12], this architecture addresses how such modular, pre-decomposed applications can be effectively coordinated and executed across a cloud-edge continuum. This aligns with the broader trend of leveraging smaller executable blocks for flexible distributed placement [2]. In this setting, the main challenge is ensuring that data flows correctly across these distributed stages, which may span different network nodes.”

To enable this execution model, we design a lightweight coordination framework for decomposed applications in the cloud-edge continuum. As shown in Figure 1, the framework is organized around an orchestrator, a set of SYS nodes, and Wasm-based execution components. Below, we explain each of these components:

**Orchestrator:** Implemented in Rust and compiled to Wasm, it tracks the registry of active SYS nodes and receives execution demands as function chain representations. It applies a latency-based routing logic, selecting nodes with the lowest round-trip time to the data source, to update the global routing table’s `assignments` field. While this routing logic can be extended with advanced placement policies, this work deliberately focuses on the fundamental deployment, termination, and reconfiguration mechanisms, leaving alternative optimization strategies for future work.

**SYS Nodes:** Operating on physical servers, these Rust-based daemons instantiate or terminate local Wasm binaries by monitoring the routing table’s `assignments` field. When a local function is launched, the SYS node registers its network endpoint in the table. In parallel, nodes continuously monitor and forward hardware metrics (CPU, memory, network) to the orchestrator, acting as the execution substrate that materializes placement decisions.

**Global Routing Table:** A shared JSON structure used to coordinate function placement across the continuum. It contains three primary fields: `routing` (mapping function identifiers to their current IP/port endpoints), `nodes` (tracking active infrastructure hosts), and `assignments` (specifying which functions must execute on each node). Initially generated by the orchestrator, this global view is replicated locally by each

SYS node to drive autonomous instantiation decisions, with local updates propagated back for global synchronization.

As illustrated in Figure 2, the WebAssembly Component Model abstracts low-level memory sharing and language-specific Application Binary Interfaces (ABIs) by utilizing a centralized WIT specification. Toolchains such as *wit-bindgen* parse this language-agnostic interface description to automatically generate the necessary glue code (*bindings*) for both the caller and the implementer domains. In this concrete scenario, a receiver module written in Rust can seamlessly invoke a function implemented in C/C++ as a native function call. The Canonical ABI handles complex data types (such as passing pixel records via WIT list interfaces) and manages memory allocations transparently across polyglot components. While passing structured lists across boundary layers prioritizes developer-friendly interoperability over zero-copy memory sharing, it eliminates the need for manual serialization or custom runtime extensions. Driven by this standard, our framework concrete execution blocks are defined as follows:

- **Receiver:** Implemented in Rust and compiled to WebAssembly, this component acts as the network plane for the application. It handles low-level socket communication, parses incoming message headers containing the execution flow sequence, and enforces routing rules. By leveraging the Wasm Component Model semantics explained above, the Receiver exposes a standardized interface defined via WIT files, allowing it to dynamically compose with and invoke the application Kernel while seamlessly shielding the service logic from network management concerns.
- **Kernel:** Implements the core business logic and is the only component written by the application developer, supportable in any programming language that compiles to WebAssembly. Through the shared WIT specification, the Kernel is composed alongside the Receiver into a single, cohesive Wasm binary. The Receiver interacts with the Kernel strictly through this language-agnostic interface to exchange payload data transparently, enabling developers to focus exclusively on functionality while the framework handles distributed chaining.

The orchestrator maps a function chain to nodes based on latency, updating a global routing table. Each node monitors its local table copy to instantiate or terminate functions, reporting its endpoints back to the orchestrator for global synchronization. Consequently, nodes can autonomously route flows to the next stage, even across different physical hosts. During execution, the Receiver component invokes the local Kernel or forwards data based on these routing rules. This architecture abstracts placement and communication, allowing a multi-stage pipeline to span multiple nodes transparently to the developer.

### IV. DEPLOYMENT AND EVALUATION

This chapter evaluates how decomposed applications executed through a WebAssembly-based runtime behave under different deployment conditions in the cloud-edge continuum.

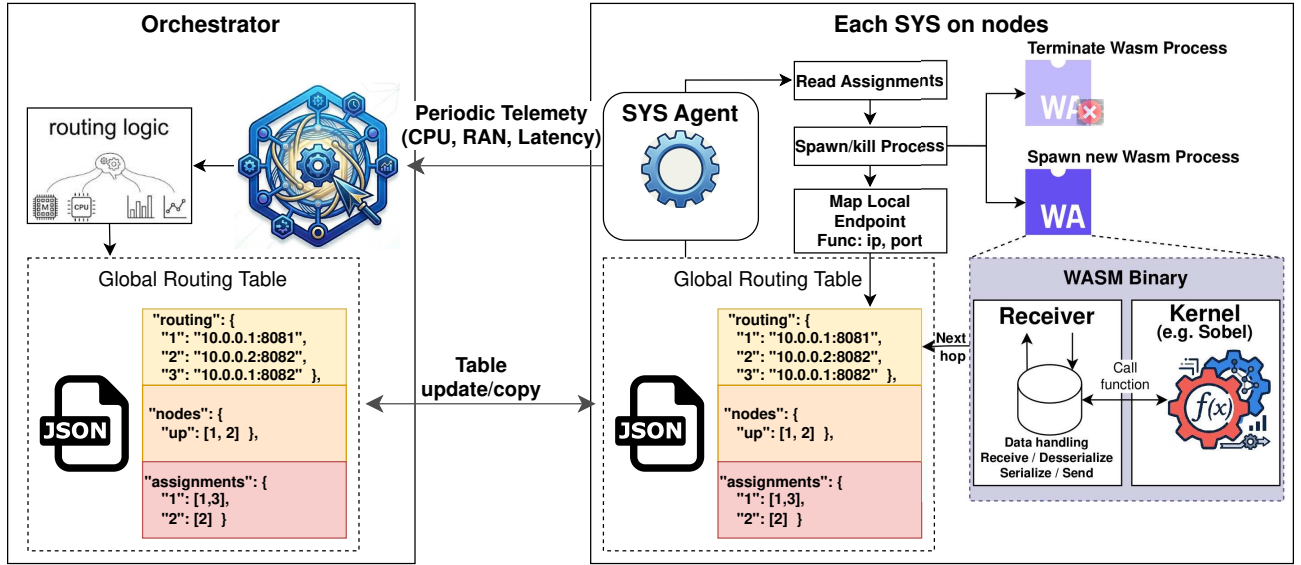


Fig. 1: Workflow example with three functions: functions 1 and 3 are placed on Node 1 and function 2 is placed on Node 2.

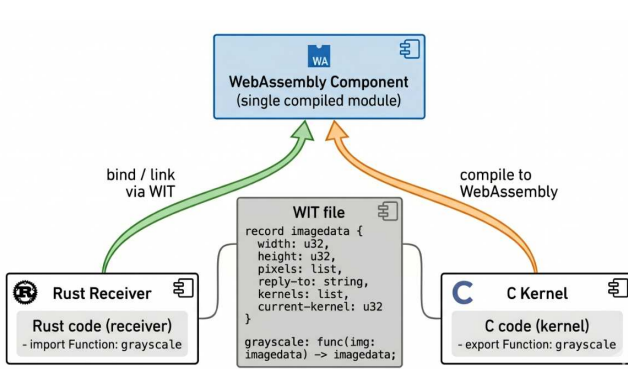


Fig. 2: Component Model Example.

The workload used in the evaluation is an image-processing pipeline composed of three computational stages: grayscale, sobel, and invert. Although this is a toy example in terms of application complexity, it accurately captures the execution structure of many future cloud-edge continuum workloads, since the output of each stage becomes the input to the next one. In other words, the relevance of this workload does not come from the sophistication of the individual operations, but from the fact that it represents a sequence of interconnected kernels that must be executed in order and potentially distributed across different nodes. This makes it a useful proxy for emerging cloud-edge scenarios in which applications are decomposed into smaller functional blocks that are steered across a heterogeneous continuum to satisfy locality, latency, or resource constraints [2].

To make the comparison more meaningful, the same test matrix was repeated for three input sizes:  $224 \times 224$ ,  $1024 \times 1024$ , and  $3840 \times 2160$ . These sizes represent progressively heavier workloads, from a CNN-like input to HD and 4K

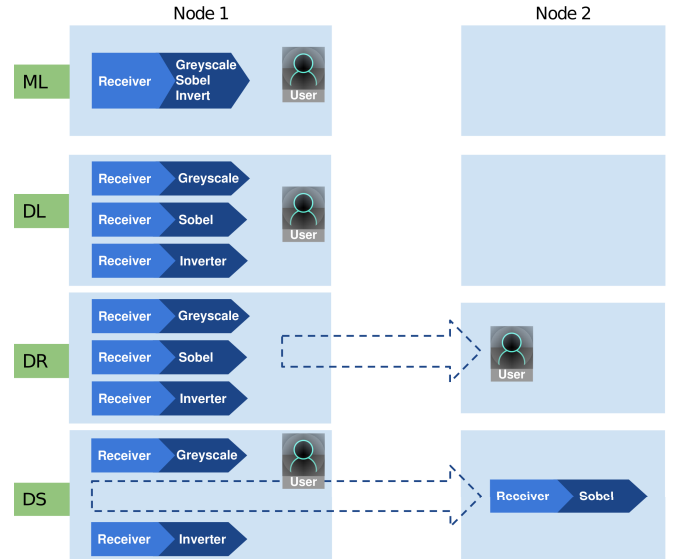


Fig. 3: Implementation scenarios.

images, and they help illustrate how the same stage-to-stage execution pattern behaves under increasing computational demand. Combined with the fact that the output of one stage becomes the input of the next, this variation in input size reinforces the pipeline nature of the workload while broadening the evaluation beyond a single toy case.

#### A. Workload models

The prototype is built on the WebAssembly component model, using WIT interfaces to compose each application from reusable components. In this setup, a Receiver component is linked with one or more application components to form a single Wasm binary, which can then be deployed either as one monolithic unit or as separate stages in a distributed pipeline.

This organization was chosen to provide a controlled comparison between local execution and distributed execution, while isolating the cost introduced by decomposing the workload and forwarding data across the network.

- **In the monolithic model**, a single Receiver is linked with one C file containing all three functions, so the three stages are compiled into one WebAssembly binary. In practice, the Receiver invokes the three stages in sequence within the same process, without any network hop between them. This model serves as a baseline for measuring the cost of executing the full pipeline locally, without distribution overhead.
- **In the distributed model**, each stage is separated into its own WebAssembly binary. As a result, the same logical application is now represented as three independent binaries, each responsible for one step in the pipeline. The output of one stage must therefore be forwarded over the network to the next stage, according to the routing information maintained by the system. This model captures the combined impact of decomposition and network distribution, which is the core behavior under study in this work.

### B. Test scenarios

The experiments were designed to isolate two effects: the overhead introduced by decomposing the application and the additional cost of moving data across nodes. While inter-node latency is inherently relevant in distributed systems, it is not the primary variable under study in this work. Instead, we aim to quantify the overhead of decomposition and the communication cost introduced by our solution in a controlled setting. To that end, the same pipeline was executed under four placement scenarios, ranging from fully local execution to partial distribution across two machines, as shown in Figure 3. This setup enables a fair comparison between monolithic execution, distributed execution without network distance, fully remote execution, and a mixed case in which only one stage is offloaded.

The experimental environment consisted of two identical servers interconnected via a high-speed local network, exhibiting an end-to-end latency of less than 1 ms. Each node is equipped with an Intel(R) Xeon(R) E5-2658 v3 CPU (32 logical cores, 2.10 GHz) and 64 GB of DDR4 RAM, organized in a dual-node NUMA architecture to ensure memory affinity and performance consistency. The software stack is based on Ubuntu 24.04 LTS (Kernel 6.8.0). To ensure lightweight and secure execution, all workloads were deployed using the Wasmtime runtime with the WebAssembly System Interface (WASI) [13]. This setup establishes a controlled baseline for performance characterization, allowing future work to explore high-latency deployments and deeper orchestrator integration.”

The following scenarios were evaluated:

- **ML:** Monolithic Local, where the full pipeline runs locally as a single WebAssembly binary (measures the baseline cost of local execution)

- **DL:** Distributed Local, where the three stages are split into separate binaries but still run on the same machine (isolates the overhead of decomposition without network distance between stages).
- **DR:** Distributed Remote, where the distributed pipeline is executed with the client on a second machine (measures the cost of full remote execution).
- **DS:** Distributed Split, where the second stage, *sobel*, is placed on the second machine while the remaining stages stay local (represents a more realistic offloading case, in which only the heaviest stage is moved to another node).

### C. Measurement method

The main metric used in this evaluation is time to complete, measured on the client side. It captures the full end-to-end execution time of the application chain, including local computation, forwarding between stages, and any overhead introduced by the distributed setup. Because the stages are chained, this metric directly reflects how effectively the system executes a decomposed workload in the continuum.

Each experiment was repeated 30 times, and the median completion time was used for comparison. For each input size, the application was executed under each scenario, producing results that can be directly compared across workload size and deployment mode, such as 4K-ML, 4K-DL, 4K-DR, and 4K-DS.

This setup reflects the type of workloads expected in future cloud-edge continuum systems, where machine-learning and image-processing pipelines are unlikely to remain monolithic. Instead, they are expected to be split into smaller kernels that can be executed across heterogeneous nodes, with each stage potentially running on a different node depending on latency and resource availability.

## V. RESULTS

This section analyzes the end-to-end completion time of the four execution scenarios across the three input resolutions, as shown in Figure 4. As expected, moving from the monolithic baseline (ML) to the distributed local model (DL) increases completion time in all scenarios. Although the overhead is smaller in absolute terms for workloads with less data, it represents a larger fraction of the total runtime; for example, in the smallest input size, the overhead accounts for approximately 30% of the total execution time, compared to 14% in the largest input size. This makes the trade-off more visible for shorter executions, while the relative impact becomes smaller as the workload grows. This result reflects the expected cost of decomposition and component interaction, which serves here as a baseline effect rather than the main focus of the work.

The difference between the distributed local case (DL) and the remotely distributed cases (DR and DS) is small across all workloads. For the HD input, DL completes in 1.996 s, while DR and DS complete in 2.025 s and 2.023 s, respectively. For the 4K workload, the corresponding values are 15.544 s, 15.657 s, and 15.726 s. These results indicate that once the cost of decomposing the pipeline is paid, placing the stages on

## Comparative Performance Analysis by Resolution

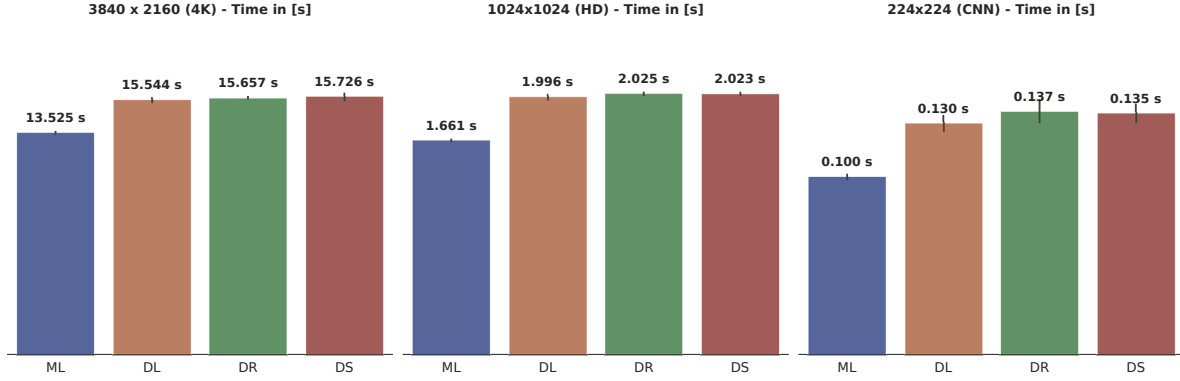


Fig. 4: End-to-end completion time across the ML, DL, DR, and DS execution scenarios for three input resolutions.

different nodes adds only a marginal additional penalty under the tested conditions.

More importantly, DR and DS behave almost identically. This shows that moving only the heavy Sobel stage to a remote node does not introduce a meaningful overhead beyond the distributed local case, which means the system can benefit from remote placement without effectively paying an extra distribution cost. In other words, the runtime can support latency-driven steering and placement across the cloud-edge continuum while keeping the additional cost of network distribution very small in a low-latency environment.

## VI. CONCLUSIONS

This work introduces a lightweight runtime framework for executing decomposed applications across the cloud-edge continuum, acknowledging that while decomposition adds overhead compared to monolithic systems, it is essential for modern distributed architectures. The main contribution lies in demonstrating that traffic can be effectively steered across application stages with controlled and predictable behavior. Experimental results show that once the initial decomposition cost is accepted, fine-grained placement and selective offloading can be executed with minimal additional performance penalties, establishing the framework as a viable coordination substrate.

While this evaluation confirms the feasibility of traffic steering under low-latency conditions, evaluating performance across higher-latency WANs and heterogeneous hardware remain essential next steps. Consequently, future work will focus on benchmarking against traditional container runtimes, optimizing WIT data structures (e.g., shared-memory abstractions), and evaluating the framework across diverse hardware ISAs (e.g., ARM edge nodes and x86 cloud servers).

## ACKNOWLEDGMENTS

This work was supported by Ericsson Telecomunicações Ltda., and by the Sao Paulo Research Foundation (FAPESP), grant 2021/00199-8, CPE SMARTNESS. This research was partially funded by CAPES, Brazil - Finance Code 001.

## REFERENCES

- [1] F.-J. Grosch, D. Dasari, N. Pereira, and A. Rowe, "Building reliable distributed edge-cloud applications with webassembly," 2022.
- [2] G. Ananthanarayanan, M. Balkwill, X. Foukas, Z. Lai, B. Radunovic, C. Settle, and Y. Zhang, "Distributed ai platform for the 6g ran," in *Proceedings of the 2nd ACM Workshop on Open and AI RAN*, ser. OpenRan '25. New York, NY, USA: ACM, 2025, p. 15–21. [Online]. Available: <https://doi.org/10.1145/3737900.3770167>
- [3] G. M. Vieira de Matos, W. R. Dias da Silva, F. L. Verdi, and A. Williams, "Decomposer: Functional decomposition and distributed execution of monolithic applications to heterogeneous resources," in *NOMS 2025-2025 IEEE Network Operations and Management Symposium*, 2025, pp. 1–10.
- [4] A. Rossberg, "WebAssembly Core Specification," 2026. [Online]. Available: <https://www.w3.org/TR/wasm-core-2/>
- [5] E. Tinto and T. Vardanega, "A runtime infrastructure for the continuum of computing," in *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 401–404. [Online]. Available: <https://doi.org/10.1145/3625549.3658832>
- [6] M. Nurul-Hoque and K. A. Harras, "Nomad: Cross-platform computational offloading and migration in femtoclouds using webassembly," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2021, pp. 168–178.
- [7] S. E. Khelifa, M. Bagaa, O. Bekkouch, M. Ahmed Ouameur, and A. Ksentini, "Extending webassembly for deep-learning inference across the cloud continuum," *IEEE Transactions on Network and Service Management*, vol. 22, no. 6, pp. 5658–5674, 2025.
- [8] ByteCodeAlliance, "An overview of wit," 2026, accessed: 2025-03-27. [Online]. Available: <https://component-model.bytecodealliance.org/design/wit.html>
- [9] C. N. C. Foundation, "Open cluster management." [Online]. Available: <https://open-cluster-management.io/>
- [10] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: end-to-end optimization stack for deep learning," *arXiv preprint arXiv:1802.04799*, vol. 11, no. 2018, p. 20, 2018.
- [11] J. Bachmeier, V. Yussupov, J. Hen, and H. Koziol, "Webassembly with wasi-nn for edge machine learning inference: Experiences and lessons learned," in *Software Architecture: 19th European Conference, ECSA 2025, Limassol, Cyprus, September 15–19, 2025, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2025, p. 343–359.
- [12] W. R. Silva, F. L. Verdi, G. M. Matos, and A. Williams, "Delivering augmented reality to the edge: An approach toward object recognition through the in-network computing paradigm," in *Proceedings of the CoNEXT on Student Workshop 2024*, 2024, pp. 13–14.
- [13] D. G. et al., "Webassembly/wasi: v0.2.0," Apr. 2024. [Online]. Available: <https://doi.org/10.5281/zenodo.11062378>