

Heterogeneous Application Orchestration in Cyber-Physical Systems

Mehmet Cihan Sakman^{1,4}, Valerio Schiavoni¹, Ronny Seiger²,
Olaf Zimmermann³, Josef Spillner⁴

¹ University of Neuchâtel, Switzerland, first.last@unine.ch

² RWTH Aachen University, Germany, first.last@rwth-aachen.de

³ University of St.Gallen, Switzerland, first.last@unisg.ch

⁴ ZHAW, Switzerland, first.last@zhaw.ch

Abstract. Cyber-physical systems (CPS) operate across a computing continuum of heterogeneous devices with varying support for execution formats such as containers, Wasm, and native binaries. While Kubernetes is the de facto orchestration standard, its container-centric model and operational overhead make it unsuitable for resource-constrained embedded devices in CPS deployments. We present an adaptive orchestration system that treats format heterogeneity as a first-class concern, enabling distributed deployment across heterogeneous CPS environments. The system selects and places components based on device capabilities and user-specified non-functional requirements (NFRs). Through a MAPE-K control loop, local device agents continuously monitor and report constraints to a central orchestrator. Upon detecting a constraint violation, the orchestrator directs the agents to locally adapt through execution format transformations, redeployments, or component suspensions. Evaluation on a distributed image processing pipeline of five microservices demonstrates deployment initialization in < 5 minutes, rapid execution format transformation in < 3 seconds, and a stable agent memory footprint of 30–40 MB even under active load. These results establish that dynamic, NFR-driven heterogeneous orchestration can be achieved with minimal overhead in resource-constrained CPS environments.

1 Introduction

Cyber-physical systems rely on a computing continuum that spans data centres, edge gateways, and resource-constrained devices deployed at the physical edge [30]. This architectural diversity reflects the distributed nature of CPS: sensor fusion occurs near data sources to reduce latency, control loops require proximity to actuators, and high-throughput analytics builds on cloud scalability [31]. Controlling this multi-tier reality in software is challenging, as common orchestration paradigms assume a level of resource homogeneity absent in CPS.

Kubernetes has become the dominant orchestration framework, including in cyber-physical and edge-oriented deployments [7,25]. However, Kubernetes is fundamentally designed for relatively homogeneous clusters with stable resource availability and a uniform container runtime model [34,39]. CPS violate

these assumptions along two critical dimensions. First, resource-constrained and embedded devices often lack the memory, CPU capacity, or operating system support required to efficiently host container runtimes and the Kubernetes node stack [20]. Second, CPS deployments operate under strict non-functional requirements (NFRs) that extend beyond performance metrics, including energy budgets on battery-powered devices, thermal constraints on passively cooled nodes, fuel levels and availability guarantees for safety-critical or remote components [11]. While Kubernetes supports basic resource limits and node affinity, it lacks support for sustainability constraints such as preferring devices with stable power supplies, avoiding overheating on passively-cooled devices, and prioritizing critical workloads during resource shortages [24].

In response, developers often adopt a pragmatic workaround: deploy all components as containers on a single host, effectively using Kubernetes within its intended homogeneous, resource-stable operating model. This centralised approach is operationally simple but forfeits benefits such as distributed computing, reduced network traffic, increased fault tolerance, and reduced total energy consumption with heterogeneous power profiles. The irony is that devices best suited for sustainable operation (solar-powered edge devices) are underutilised because orchestration tools struggle to reason about non-functional constraints such as energy availability, thermal limits, or intermittent connectivity [24,26].

The core obstacle is the combination of execution-format homogeneity and limited awareness of non-functional constraints in current orchestration systems. Many constrained devices can execute lightweight workloads when appropriate formats are used. Wasm runtimes, for example, are designed for fast startup and low memory footprints, making them suitable for environments where container-based execution is impractical [14,21]. Similarly, native binaries can be deployed on capable nodes to avoid runtime overhead. Both execution formats and device-level constraints are well understood; what is missing is an orchestration layer that treats execution-format heterogeneity and NFRs as first-class concerns.

Moreover, CPS environments are inherently dynamic. Energy availability, thermal conditions, device health, and workload characteristics evolve over time, invalidating static placement decisions made at deployment. Effective CPS orchestration therefore requires continuous monitoring and constraint-aware adaptation, allowing workloads to be redistributed or execution formats adjusted as devices approach operational limits. Such feedback-driven control aligns with established autonomic management models like MAPE-K loops [18].

To align with these concerns, this paper makes the following contributions:

- An NFR-driven adaptive orchestration framework for heterogeneous CPS environments, where device capabilities, sustainability-related NFRs, and runtime format support are jointly considered during distributed deployment.
- A runtime-adaptive orchestration system that performs constraint-aware component placement and reconfiguration using MAPE-K control loops to maintain system stability.
- A semi-automated Wasm component builder that bridges the gap between stateful Python development and stateless Wasm execution. This tool au-

- tomatoes the extraction and compilation of computational logic into Wasm binaries, eliminating the need for manual refactoring.
- Demonstration of the system on a heterogeneous edge-to-cloud testbed running a distributed image processing pipeline. Our evaluation confirms that NFR-driven adaptation—specifically dynamic format switching—reduces artefact storage overhead by up to 75% and enables sub-3-second reconfiguration latencies under constraint violations.

The paper is structured as follows: Section 2 provides background on CPS environments, orchestration limitations, relevant execution formats, and NFRs. Section 3 discusses related work. Section 4 details our system design and architecturally significant requirements. Section 5 describes the prototype implementation, followed by the experimental evaluation in Sec. 6. Section 7 summarizes and concludes.

2 Background

To contextualize the need for adaptive orchestration, this section outlines the CPS computing continuum, the limitations of container-centric approaches, and the dynamic NFRs that necessitate heterogeneous execution formats.

2.1 The CPS Computing Continuum

CPS span a continuum from cloud infrastructure to resource-constrained IoT devices. The heterogeneous layering reflects physical reality: sensors co-located with monitored processes, edge nodes enabling low-latency control loops, cloud resources providing scalable analytics. Cloud nodes provide multi-core x86_64 processors with tens of gigabytes of memory, edge gateways use ARM SoCs with 1–4 GB RAM, and embedded devices have 256–512 MB memory with single-core CPUs clocked at hundreds of MHz. Power profiles range from stable data center supplies to battery power and energy harvesting. Network connectivity spans high-bandwidth wired connections to constrained wireless protocols.

2.2 Kubernetes and the Homogeneity Assumption

Kubernetes abstracts infrastructure as clusters of nodes capable of executing containerized workloads under a unified control plane. This abstraction works well in homogeneous servers. When extended to CPS, it exposes several fundamental mismatches between Kubernetes’ design assumptions and CPS realities.

Resource requirements. Its node agent (kubelet) maintains continuous communication with the control plane and manages container lifecycles, requiring a non-trivial resource footprint [39]. While lightweight Kubernetes distributions such as K3s or K0s reduce control-plane binary size and initial memory requirements, the operational overhead remains significant on many edge-deployed devices [37]. In addition, container runtimes introduce further memory and storage

overhead through image management, background daemons, and filesystem layers. On constrained devices, this baseline consumption significantly reduces the capacity available for application workloads, limiting deployment feasibility.

Container runtime overhead. Containers obtain namespaced isolation through cgroups, assuming a fully-featured Linux kernel. While efforts such as K3s and KubeEdge [1] reduce the footprint, they cannot eliminate the fundamental runtime overhead with layering, union filesystems, and daemon processes [39].

Operational complexity. Kubernetes assumes reliable connectivity, sufficient storage for images, and rolling update capability. Embedded devices lack these requirements. These are not flaws but consequences of design choices optimized for Kubernetes’ primary use case which is a subset of the CPS continuum.

2.3 Heterogeneous Execution Formats

CPS require support for multiple execution formats to accommodate device diversity. We identify three format classes. *Containers* provide process isolation through OS-level virtualization, mature tooling and vast ecosystems. Implementations include Docker, Podman, and LXC [4]. Despite overhead, containers remain the preferred choice for servers and capable edge gateways with sufficient memory and storage. *WebAssembly (Wasm)* serves as lightweight, portable alternative. Runtimes like WasmEdge [36] and Wasmtime [6] require minimal memory (often < 50 MB), start in milliseconds, and provide sandboxing without OS support [13,22,38], making them suitable for constrained devices where containers exceed resources. However, the Wasm ecosystem is less mature. *Native execution* directly on the host OS eliminates the overhead and simplifies hardware access, though it sacrifices portability and isolation.

These execution formats are *complementary*, not competitive. A single CPS application might appropriately use containers on edge gateways, Wasm on moderately constrained sensors, and native binaries on specialized microcontrollers. Current orchestration frameworks, however, assume format homogeneity and lack mechanisms to reason about format-specific constraints.

2.4 Non-Functional Requirements (NFRs) and Constraints in CPS

CPS deployments must satisfy operational constraints beyond functional correctness. We identify critical NFR dimensions following the ISO/IEC 25010 quality model [15], extended with sustainability as a distinct category following the Karlskrona Manifesto’s principles for environmental and economic considerations [3].

Energy Efficiency and Sustainability. Battery-powered or energy-harvesting devices operate within explicit power budgets. Solar-powered nodes must adapt workloads to energy availability; violating constraints risks device shutdown or reduced operational lifetime. Energy efficiency has emerged as a prominent sustainability concern in distributed systems [2,27,32], yet existing orchestration

approaches operate within single execution formats and do not consider format heterogeneity as an adaptation mechanism.

Thermal Constraints. Passively-cooled sensors have maximum operating temperatures where compute-intensive workloads can cause overheating, leading to throttling, errors, or hardware damage. Thermal management is a physical constraint that must be actively monitored and enforced through orchestration.

Criticality and Availability. Applications have varying criticality levels: control loops governing physical safety must maintain continuous operation (high availability), while monitoring or logging services can tolerate degradation during resource contention. This criticality-based prioritization aligns with reliability requirements in safety-critical CPS.

These constraints are inherently dynamic: energy availability fluctuates, ambient temperature affects thermal capacity, and workloads shift unpredictably. Kubernetes alone provides CPU and memory limits and basic placement controls, but cannot react to thermal stress, adapt to renewable energy cycles, or suspend lower-priority workloads when critical services require resources. NFR satisfaction can benefit from execution formats: a service violating thermal or energy constraints, as a container may operate within limits as a Wasm module, enabling deployment where container execution would be infeasible.

3 Related Work

Container and Edge Orchestration. Kubernetes and its lightweight and edge variants such as K3s and KubeEdge provide mature container orchestration with custom-metric scheduling and partial NFR-aware placement [39,33]. However, these systems primarily optimize placement decisions within a container-centric control plane and rely on static packaging assumptions. Hybrid edge designs such as [17] explore hybrid virtualization (combining containers and unikernels) to improve resource utilization on constrained IoT nodes. Their evaluation, however, is restricted to specific hardware classes and focuses on design-time architectural trade-offs rather than continuous runtime adaptation. Likewise, autonomous energy-aware schedulers for serverless edge computing [32,2] employ MAPE-K control loops to monitor battery-powered nodes, yet dynamically offload tasks based on energy availability zones, without considering execution format transition or runtime optimization.

Declarative Modeling and TOSCA-Based Approaches. TOSCA [23] enables declarative specification of application topologies and NFRs. Platforms such as OpenTOSCA [5] and Cloudify [8] support automated deployment and policy-driven orchestration, including extensions for edge and IoT scenarios. These approaches emphasize model-driven provisioning and constraint-aware placement but typically operate at deployment time, with limited support for closed-loop adaptation. Furthermore, policy enforcement is usually limited to placement and scaling decisions, without integrating dynamic execution-environment management.

WebAssembly in Distributed and Serverless Systems. Systems such as wasm-Cloud [35] and Fermion [12] enable distributed Wasm deployment, while Krustlet [10] integrates WASI workloads into Kubernetes by replacing the kubelet and executing modules without relying on a traditional container runtime. However, the Krustlet project is no longer actively maintained and has largely been superseded by containerd-based approaches such as runwasi [9], which embed Wasm runtimes as shims within containerd to allow coexistence with containers under a unified control plane. Authors in [19] explore Wasm-native orchestration as an extension to Kubernetes, demonstrating Wasm as a portable and efficient execution substrate while remaining tied to container-centric or serverless control planes. In contrast, we treat Wasm and native binaries as first-class orchestration targets that can bypass container runtimes entirely, reducing per-node footprint and runtime overhead. Additionally, we support automated transformation from source components to executable Wasm artifacts, facilitating seamless adoption of lightweight execution formats in heterogeneous edge deployments.

4 System Design

This section details the architecture and operational mechanics of the proposed orchestration framework, progressing from core design objectives to system components, declarative configuration, and the runtime adaptation loop.

4.1 Design Objectives

Addressing the architectural limitations and CPS requirements outlined in Section 2.1, we derive four Architecturally Significant Requirements (ASRs):

- **R1: Heterogeneity Support.** The system must support multiple execution formats (Containers, Wasm, Native) rather than assuming a uniform runtime.
- **R2: Minimal Orchestration Overhead.** The agents must operate within strict embedded resource limits (e.g., < 50 MB idle memory) to maximize capacity for application workloads.
- **R3: NFR-Driven Adaptation.** The system must evaluate and react to physical constraints (e.g., energy, thermal) beyond standard computational limits (CPU/RAM).
- **R4: Dynamic Lifecycle Management.** The system must continuously react to sensor measurements and adapt system state to ensure workloads remain active and available.

To fulfill these ASRs, our autonomic orchestration framework employs a MAPE-K control loop to manage multi-format execution (Container, Wasm, Native) alongside dynamic service placement and lifecycle management in heterogeneous CPS environments.

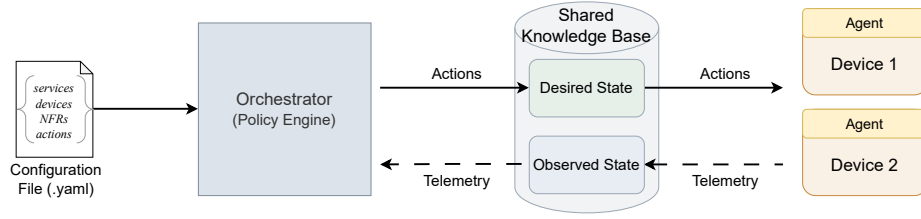


Fig. 1: Conceptual architecture of the NFR-driven orchestration system.

4.2 Architecture Overview

The system comprises three components (cf. Fig. 1): a central orchestrator, agents, and a shared knowledge base. While the implementation focuses on Containers, Wasm, and Native processes, the architectural separation between planning and enactment allows additional execution backends to be integrated through agent-side runtime adapters. The orchestrator reads deployment specifications, evaluates constraint rules, and issues adaptation commands. Agents monitor and report local resources, and execute deployment actions. The shared knowledge base serves as the coordination backbone and implements the Knowledge (K) component of the MAPE-K loop. It maintains two abstract views: an *observed state* reflecting telemetry from agents and a *desired state* representing deployment rules and adaptation decisions computed by the orchestrator.

Users specify deployments via a YAML configuration defining services (multi-format microservices), devices (resource profiles and supported formats), constraints (operational thresholds), and actions (rule-based policies). The orchestrator prepares all required artifacts, multi-architecture container images and Wasm components, before deployment, distributing them to registered devices. This pre-deployment phase eliminates transfer delays at runtime.

Communication is database-centric for simplicity, though the architecture supports alternative coordination mechanisms (e.g., message queues or REST APIs). Agents poll the database for commands and write status updates, while the orchestrator evaluates observed state and issues adaptation directives.

4.3 Configuration Model

Deployment specifications follow a declarative model capturing devices, services, constraints, and adaptation policies as depicted in Fig. 2. Constraint thresholds are expressed as percentages (CPU $\mathbb{C}$, memory $\mathbb{M}$ & battery $\mathbb{B}$ utilization) or absolute values (temperature, power consumption, network latency). These constraints span multiple NFR dimensions: $\mathbb{C}$ and $\mathbb{M}$ utilization reflect performance efficiency, $\mathbb{B}$ and power consumption address energy sustainability, and temperature represents thermal management for passively-cooled embedded devices where sustained computation can cause overheating and hardware damage. The system supports three adaptation actions: `MIGRATE` relocates services to

<pre> devices: - name: edge cpu_cores: 4 memory_mb: 4096 formats: [container, wasm] static_ip: 10.10.100.1 platform: linux/arm64 - name: sensor cpu_cores: 1 memory_mb: 512 formats: [wasm, native] static_ip: 10.20.100.1 platform: linux/arm64 - name: cloud cpu_cores: 16 memory_mb: 32768 formats: [container] static_ip: 192.168.1.100 platform: linux/amd64 </pre>	<pre> services: - name: image-preprocessor preferred: [sensor, edge, cloud] formats: container: image: repo/preprocessor build: ./services/pre wasm: wit: ./pre/interface.wit impl: ./services/pre/worker.py constraints: cpu: critical: 90.0 medium: 70.0 healthy: 50.0 actions: cpu: medium: [TRANSFORM] critical: [MIGRATE] </pre>
--	---

(a) Device profiles defining resources.

(b) Service constraints and formats.

Fig. 2: View of the unified configuration.yaml file presenting device profiles (a) and service specifications (b).

different devices maintaining format where supported, TRANSFORM switches execution format on the same device (container $\leftrightarrow$ Wasm), and PAUSE suspends non-critical services. The preferred list defines initial placement order; future work will incorporate intelligent device selection.

4.4 MAPE-K Control Loop and Adaptation

The system implements a MAPE-K control loop [18] for heterogeneous runtime orchestration (cf. Fig. 3) addressing dynamic lifecycle management (R4). Agents continuously Monitor resource metrics ($\mathbb{C}$, $\mathbb{M}$, $\mathbb{B}$, temperature) at configurable intervals, writing measurements to the shared database (Knowledge). The orchestrator polls for agent reports, Analyzing metrics against declarative constraint thresholds and violation persistence conditions. Violations trigger adaptation Planning; the orchestrator identifies affected services, evaluates permitted actions, and selects candidate devices for migration or format transformation. MIGRATE relocates a service to a different device from the preferred list that satisfies constraints and supports the service’s format; if the target supports only a different format, migration implicitly includes transformation. TRANSFORM switches execution format on the same device (e.g., container $\rightarrow$ Wasm to reduce memory footprint), requiring brief service interruption. PAUSE suspends execution when neither migration nor transformation can satisfy constraints; if no valid action succeeds, the orchestrator logs a warning and stops the service to protect devices. When multiple actions are applicable for the same violation, the orchestrator resolves them according to the left-to-right priority order defined in the configuration, ensuring deterministic adaptation and preventing conflicting enactment requests for the same service.

Agents poll the database for commands and `Execute` locally: managing service lifecycles and format switches. The shared database maintains system state across device capabilities (*devices*), service deployments (*services*), constraints (*constraints*), and adaptation history (*actions*). Agent heartbeats enable failure detection; missing updates trigger device offline status and service migration. To prevent thrashing and overreaction to transient spikes, the system implements hysteresis: constraints must remain violated for a configurable number of consecutive monitoring cycles before triggering adaptation, and cooldown periods prevent rapid re-adaptation after enactment. When constraints return to healthy ranges, the orchestrator restores the initial service placement defined in *preferred*, ensuring a predictable deployment topology.

4.5 Multi-Format Artifact Management

To satisfy the format heterogeneity (R1), the orchestrator prepares execution artifacts for all currently supported service formats defined in the configuration before deployment. For containers, `Buildx` generates multi-architecture images supporting `x86_64` and `ARM` platforms, ensuring compatibility across the device continuum. For `Wasm`, the system automates component generation from Python source code.

The `Wasm` build pipeline extracts pure computational logic from standard Python workers using static analysis to remove database connections, networking code, and control loops. The extracted logic is wrapped in a `Component Model v2` adapter matching the service’s `.wit` interface definition. The adapter is then compiled using `componentize-py` and executed via `Wasmtime`. This automation currently supports services with cleanly structured code separating business logic from infrastructure concerns; future work will extend to additional languages and handle more complex code patterns for the `Wasm` format.

During the deployment initialization phase, pre-built artifacts are distributed to all candidate devices listed in *preferred*. Agents store artifacts locally, enabling sub-second format switching during `TRANSFORM` operations without network transfer overhead. Native execution, when specified, are assumed user-provided and deployed similarly.

5 Implementation

This section details our open-source implementation [28]. Fig. 3 illustrates the end-to-end operational pipeline, capturing both artifact provisioning and run-time coordination between the central orchestrator and distributed agents.

5.1 System Components

The orchestrator executes on a designated coordination host with access to a shared SQL database and a local container registry (code available [28]). Upon boot, the orchestrator parses the `configuration.yaml` manifest to extract

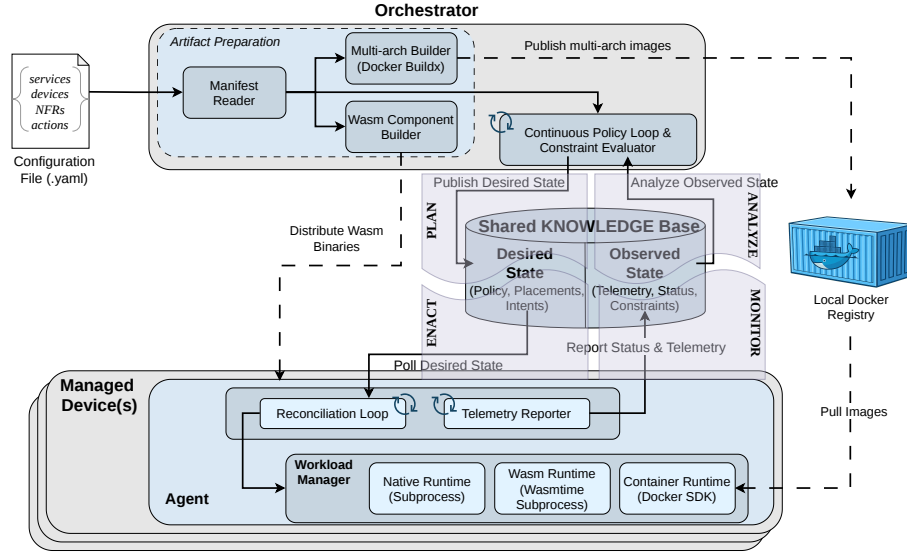


Fig. 3: System architecture implementing the MAPE-K cycle. The Orchestrator handles the *Analyze & Plan* phases alongside artifact preparation. The Knowledge Base synchronizes state with Devices, where Agents perform *Enact* (reconciliation) & *Monitor* (telemetry) operations.

service definitions, device capability profiles, supported execution formats, placement preferences, and constraint policies. Based on the declared formats, it first builds all required multi-arch container images and populates the local registry. If any candidate device supports Wasm execution, a dedicated automation utility compiles the corresponding service implementations into portable Wasm component binaries and distributes them to the supported devices.

After artifact preparation, the orchestrator computes initial service-to-device assignments based on placement priorities and capabilities, writing them to the shared database as the desired state. It then transitions into continuous analysis mode. The constraint evaluator parses NFRs as thresholds and adaptation actions. Orchestrator evaluates these against the observed state and updates the desired state, fulfilling the requirement for NFR-driven adaptation (R3).

Agents run as containers on each managed device. Each agent includes a telemetry reporter that periodically measures abstract device state and writes it to the shared database as observed state. The agent’s primary reconciliation loop polls the desired state and manages the local service lifecycle accordingly. The agent container exhibits an idle memory footprint of approximately 30–40 MB, excluding managed services, satisfying the requirement for minimal orchestration overhead (R2). Execution format management varies by runtime but remains encapsulated within the agent. For container-based services, agents use the Docker Python SDK to pull images from the local registry, create or termi-

nate containers, and inspect runtime status. For Wasm services, agents launch the Wasmtime runtime as a subprocess, execute the compiled component binary, and manage it via process control mechanisms. For native execution, agents spawn the executable directly as a subprocess with appropriate environment variables, managing it through standard process signals without intermediary runtimes. Format-specific handling is abstracted from the orchestrator, allowing uniform desired-state updates independent of execution format, directly realizing the objective of heterogeneity support (R1).

5.2 Artifact Preparation Pipeline

Orchestration begins with *artifact preparation*, a pre-deployment phase transforming configurations into deployable executables. This decouples building from the runtime lifecycle, ensuring artifacts (container images, Wasm and native binaries) are locally available on candidate devices to eliminate network overhead during adaptation.

Multi-architecture container images are generated by the *Multi-arch Builder* (cf. Fig. 3). The orchestrator parses the manifest for target architectures (e.g., `linux/amd64`, `linux/arm64`) and uses Buildx to build variants in parallel. These are pushed to the local registry, enabling agents to pull hardware-specific images without on-device compilation.

Semi-Automated Wasm component builder operationalize the Python logic in restricted environment (cf. Fig. 4). This pipeline addresses a fundamental architectural mismatch: while developers rely on stateful services with direct database access, Wasm components are often treated as stateless in practice due to the immaturity of the ecosystem—specifically, the WASI-SQL interface remains a stagnant Phase 1 proposal [16]. To mitigate the need for manual refactoring, our pipeline automates the transformation of Python workers into compliant, stateless artifacts. The process executes within a reproducible Docker-based factory. First, an *extractor* parses the WebAssembly Interface Types (`.wit`) and performs static analysis to selectively isolate pure computational logic, discarding unsupported infrastructure code like database loops. Second, a *generator* synthesizes a Component Model adapter to bridge Python’s dynamic typing with Wasm’s strict interfaces. Finally, the *compilation* step runs `componentize-py` to bundle the interpreter and dependencies into a standalone binary, effectively decoupling the developer’s logic from the complexities of the Wasm toolchain.

Currently, Wasm/WASI components have limited support for native Python extensions [29], requiring developers to structure implementations using pure Python libraries compatible with the target Wasm runtime. Furthermore, function names in the source code must match the exports declared in the `.wit` interface for the *extractor* to identify valid entry points. Since components are invoked per request in our deployment model, developers must provide a host wrapper script to manage the service’s control loop (e.g., database polling), which the orchestrator launches to trigger the Wasm component at runtime.

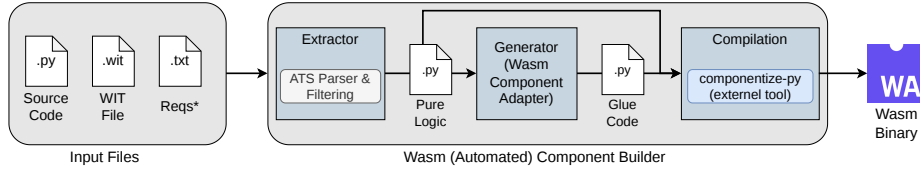


Fig. 4: Automated Wasm component generation pipeline illustrating logic extraction and artifact compilation. The asterisk (*) denotes optional inputs.

Resulting Wasm binaries range from 20–40 MB, significantly smaller than equivalent 100–150 MB container images. This reduction benefits constrained storage and accelerates adaptation: switching formats requires only starting a new subprocess rather than unpacking image layers.

6 Evaluation

We evaluate the NFR-driven adaptive orchestration system using a distributed image processing pipeline deployed across heterogeneous CPS infrastructure. This evaluation assesses whether the system satisfies the design objectives (R1–R4) in Section 4.1 by measuring initialization overhead, multi-format artifact footprint, and operational behavior under NFR-driven reconfiguration.

6.1 Experimental Setup

Testbed. We deploy a YOLOv8-based distributed image processing pipeline across three heterogeneous devices (edge-to-cloud) to validate our design objectives (Section 4.1). Its computational intensity induces the physical constraints (CPU, power) needed to evaluate NFR-driven adaptation (R3) and lifecycle management (R4). Furthermore, the architectural diversity (x86_64 and ARM64) demonstrates format heterogeneity (R1) and proves the agent operates with minimal overhead on constrained nodes (R2).

Table 1 summarizes the device specifications. All devices share access to the database hosted on the PC and network-attached storage (NAS) via NFS mounts. The PC hosts the orchestrator and a local container registry. Each device, including the PC, runs an agent that reconciles desired state and manages the local service lifecycle.

Workload and Policies. The pipeline consists of five microservices: `ingest` registers incoming data; `preprocess` normalizes images; `analyze_light` performs lightweight filtering; `analyze_heavy` executes YOLOv8 object detection; and `store_index` persists results. Table 2 details the initial placement preferences, execution formats, and constraint-driven adaptation policies for each service. To prevent oscillation, orchestrator enforces a 3-cycle (30s) violation persistence check before triggering adaptation, followed by a 60s cooldown.

Table 1: Testbed device specifications and supported execution formats.

Device	Model	CPU	Memory	Platform	Formats
PC	Core Ultra 5 135U	12 cores	32768 MB	linux/amd64	C, W, N
Edge	Raspberry Pi 400	4 cores	4096 MB	linux/arm64	C, W, N
NAS	QNAP TS-216G	4 cores	4096 MB	linux/arm64	C, N

C = Container, W = WebAssembly, N = Native

Table 2: Service deployments and threshold-based adaptation policies.

Service	Preference	Fmt	Constraint (H / M / Cr)	Actions
ingest	PC	C	—	—
preprocess	PC	C	—	—
analyze_light	Edge, PC	C, W	CPU: 50/70/90%	M:TR, Cr:MG
analyze_heavy	Edge, PC	C	Pwr: 80/-/20%	Cr:MG
store_index	NAS	C	Mem: 50/-/90%	Cr:PS

H / M / Cr = Healthy / Medium / Critical thresholds. C = Container, W = WebAssembly. TR = Transform, MG = Migrate, PS = Pause.

Artifact Sizes. Table 3 reports the uncompressed artifact sizes. Container images are stored and transferred as compressed layer archives by the local registry; consequently, the data pulled over the network is substantially smaller than the reported uncompressed size. Layers already cached on a device are skipped entirely, further reducing pull latency for subsequent deployments. Despite bundling YOLOv8n model weights, agents pulled the `analyze_heavy` image in under 100 s. For `analyze_light`, the 39 MB Wasm module represents over 75% size reduction compared to its 164 MB container counterpart, enabling rapid format switching on edge nodes.

6.2 Results

System Initialization. Full pipeline deployment completed in under five minutes from orchestrator boot to all five services reaching a running state across the three devices. As reported in Table 3, artifact sizes vary considerably: most pipeline stages occupy 164–184 MB as container images, while `analyze_heavy` reaches 1.87 GB owing to its bundled model weights. Despite this, the heaviest image was pulled over the local network in approximately 70 s, with all necessary artifacts across the pipeline retrieved in under 100 s. During this phase, the agents maintained a stable memory footprint of 30–40 MB even while parsing desired state, coordinating multi-format artifact downloads, and continuously reporting telemetry. This consistent resource profile confirms adherence to the minimal orchestration overhead requirement (R2) under idle and active load.

Manual Triggering and Timing Interpretation. To evaluate adaptation behavior in a controlled and reproducible manner, constraint violations were triggered

Table 3: Service artifact sizes (Wasm only where supported).

	ingest	preprocess	analyze_light	analyze_heavy	store_index
Container	184 MB	164 MB	164 MB	1870 MB	175 MB
Wasm	—	—	39 MB	—	—

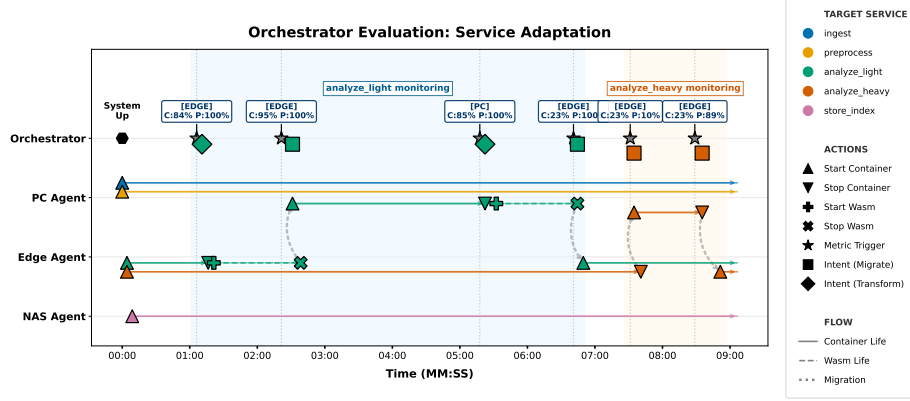


Fig. 5: Timeline of orchestration events showing service lifecycle actions, intent triggers, and telemetry-driven adaptations. Horizontal lines track execution duration, while curved arrows visualize cross-device migration flows.

manually rather than relying on organic workload-driven telemetry from the agents and represented by a star in Fig. 5. This approach decouples adaptation mechanics from workload variability and enables precise observation of system response under known, repeatable conditions. Accordingly, all reaction times reported below reflect the orchestrator’s adaptation loop latency and the time required for services to reach an operational state where they can accept work; they do not include application-level cold-start overhead such as model loading or initialization of service-specific resources.

NFR/Constraint-Driven Adaptation. Fig. 5 presents the orchestration timeline, visualizing service execution flows and migration paths alongside lifecycle events, metric triggers, and issued intents. Two services undergo constraint-driven adaptation: `analyze_light` due to CPU pressure, and `analyze_heavy` due to power depletion.

For `analyze_light`, the orchestrator records a CPU utilization ($\mathbb{C}$) of 84% on the Edge device, exceeding the 70% medium threshold defined in Table 2. After three consecutive monitoring cycles confirm the sustained violation (approximately 30 s), the orchestrator issues a TRANSFORM intent, instructing the Edge agent to replace the running container with a Wasm process on the same host. The end-to-end transition completes in 3 seconds, demonstrating rapid NFR-driven adaptation (R3) via format transformation (R1). When $\mathbb{C}$ subsequently

spikes to 95%, exceeding the 90% critical threshold, the orchestrator escalates to a `MIGRATE` intent, relocating `analyze_light` to the PC. There, $\mathbb{C}$ rises to 85%, again exceeding the threshold, triggering another `TRANSFORM` intent that converts the service from container to Wasm on the PC. Once the Edge device returns to a healthy state after the 60-second cooldown, the orchestrator migrates `analyze_light` back to Edge and restores its original container format, re-establishing the initial deployment configuration via continuous lifecycle management (R4). This demonstrates the system’s ability to not only react to constraint violations across devices but also to restore services to their preferred placement once conditions stabilize.

For `analyze_heavy`, a power depletion on the Edge device—battery level ($\mathbb{B}$) falling below the 20% critical threshold in Table 2—triggers a `MIGRATE` intent. The orchestrator moves the container to the PC. When $\mathbb{B}$ recovers to 89% and stabilizes above the healthy threshold after the 60-second cooldown, the orchestrator migrates `analyze_heavy` back to its original Edge placement, demonstrating successful constraint resolution and service restoration to the preferred deployment configuration.

7 Conclusion

We contribute an adaptive, NFR-driven orchestration framework that treats execution format heterogeneity and physical constraints (e.g., energy, thermal limits) as first-class concerns to overcome the limitations of traditional, container-centric CPS deployments. By implementing a MAPE-K control loop, the system continuously monitors device states and dynamically manages service lifecycles through in-place format transformations, component migrations, or suspensions [28]. Our evaluation on a distributed edge-to-cloud image processing pipeline successfully validated our architecturally significant requirements (R1–R4). Specifically, full system deployment—encompassing multi-architecture artifact preparation, distribution, and initial agent bootstrapping—completed in under five minutes, successfully handling complex model artifacts up to 1.87 GB. Furthermore, constraint-driven format switching between container and Web-Assembly runtimes executed in under 3 s, resolving critical resource pressure locally without requiring network-intensive migrations. Throughout these active lifecycle events, agents maintained a highly stable 30–40 MB memory footprint. These results demonstrate that dynamic, multi-format orchestration can be achieved with minimal overhead, preserving essential compute capacity on resource-constrained edge nodes. Future work will explore alternative coordination mechanisms to improve scalability; extend the agent-based architecture to support additional execution backends beyond containers, Wasm, and native processes; integrate learning-based policy engines for adaptive constraint synthesis; and broaden Wasm automation beyond Python to increase CPS applicability.

Acknowledgments. This work has received funding from the Swiss National Science Foundation under Grant No. 10002384 (*TaSSAreCt* project).

References

1. Kubernetes Native Edge Computing. <https://kubedge.io/>, accessed: 2026-02-19
2. Aslanpour, M.S., Toosi, A.N., Cheema, M.A., Gaire, R.: Energy-Aware Resource Scheduling for Serverless Edge Computing. In: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid). pp. 190–199 (2022)
3. Becker, C., Chitchyan, R., Duboc, L., et al.: Sustainability Design and Software: The Karlskrona Manifesto. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 2, pp. 467–476 (2015)
4. Ben Kebaier, N., Geib, B., Lyczkowski, E., Venanzi, R., Barth, M.: Real-Time Performance Evaluation of Containerized Virtual PLCs: A Comparative Study of Docker and Podman for Industrial Automation. In: IEEE 30th Intl Workshop on Computer Aided Modeling and Design (CAMAD). pp. 1–6 (2025)
5. Binz, T., Breitenbücher, U., Haupt, F., et al.: OpenTOSCA – A Runtime for TOSCA-based Cloud Applications. In: Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013). LNCS, vol. 8274, pp. 692–695. Springer (2013)
6. Bytecode Alliance: Wasmtime: A lightweight WebAssembly runtime. <https://wasmtime.dev> (2025), accessed: 2025-08-14
7. Cai, Z., Buyya, R.: Inverse Queuing Model-Based Feedback Control for Elastic Container Provisioning of Web Systems in Kubernetes. *IEEE Transactions on Computers* **71**(2), 337–348 (2022)
8. Cloudify Platform Ltd.: Cloudify: Open Source TOSCA-based Cloud Orchestration Platform. <https://docs.cloudify.co/> (2024), accessed: 2026-02-16
9. containerd Project: runwasi: WebAssembly Shim for containerd. <https://github.com/containerd/runwasi> (2023), accessed: 2026-02-16
10. deislabs: Krustlet: Kubernetes Kubelet for WebAssembly. <https://github.com/krustlet/krustlet> (2020), accessed: 2026-02-16
11. Derler, P., Lee, E.A., Sangiovanni Vincentelli, A.: Modeling Cyber-Physical Systems. *Proceedings of the IEEE* **100**(1), 13–28 (2012)
12. Fermyon Technologies: Spin: Developer Tool for Building WebAssembly Microservices. <https://developer.fermyon.com/spin/> (2021), accessed: 2026-02-16
13. Haas, A., Rossberg, A., Schuff, D.L., et al.: Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 185–200 (2017)
14. Has, M., Xiong, T., Abdesslem, F.B., Kušek, M.: WebAssembly on Resource-Constrained IoT Devices: Performance, Efficiency, and Portability (2025), <https://arxiv.org/abs/2512.00035>
15. ISO/IEC: ISO/IEC 25010:2011 Systems and software engineering Product quality model. International Organization for Standardization (2011), <https://www.iso.org/standard/35733.html>
16. Jiaxiao Zhou, Dan Chiarlone, D.J.: WebAssembly System Interface, SQL API (2024), <https://github.com/WebAssembly/wasi-sql>, accessed: 2026-02-16
17. Kaiser, S., Tosun, A.S., Korkmaz, T.: Edge System Design Using Containers and Unikernels for IoT Applications (2024), <https://arxiv.org/abs/2412.03032>
18. Kephart, J., Chess, D.: The vision of autonomic computing. *Computer* **36**, 41–50 (2003)
19. Kjorveziroski, V., Filiposka, S.: WebAssembly Orchestration in the Context of Serverless Computing. *Journal of Network and Systems Management* **31** (2023)

20. Kodakandla, N.: Optimizing Kubernetes for Edge Computing: Challenges and Innovative Solutions **4**, 210–221 (2021)
21. Ménétrey, J., Pasin, M., Felber, P., Schiavoni, V.: WebAssembly as a Common Layer for the Cloud-edge Continuum. In: Proc. of the 2nd Workshop on Flexible Resource and Application Management on the Edge. p. 3–8. FRAME, ACM (2022)
22. Ménétrey, J., Pasin, M., Felber, P., et al.: A comprehensive trusted runtime for WebAssembly with Intel SGX. IEEE TDSC **21**, 3562–3579 (2023)
23. OASIS: Topology and Orchestration Specification for Cloud Applications Version 1.0. Oasis standard, OASIS (2013), <http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html>
24. Orive, A., Agirre, A., Truong, H.L., et al.: Quality of Service Aware Orchestration for Cloud-Edge Continuum Applications. Sensors **22**(5) (2022)
25. Pahl, C., Lee, B.: Containers and Clusters for Edge Cloud Architectures – A Technology Review (2015)
26. Patros, P., Spillner, J., Papadopoulos, A.V., et al.: Toward Sustainable Serverless Computing. IEEE Internet Computing **25**, 42–50 (2021)
27. Rocha, I., Göttel, C., Felber, P., et al.: Heats: Heterogeneity-and energy-aware task-based scheduling. In: 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). pp. 400–405. IEEE (2019)
28. Sakman, M.C.: CPS orchestrator with distributed image processing pipeline (2026), <https://doi.org/10.5281/zenodo.18712919>
29. Sakman, M.C., Spillner, J., Schiavoni, V.: Speeding Up the Development for the Computing Continuum with WebAssembly. In: 2025 IEEE 30th Pacific Rim International Symposium on Dependable Computing (PRDC). pp. 12–22 (2025)
30. Satyanarayanan, M.: The Emergence of Edge Computing. Computer **50**(1), 30–39 (2017)
31. Shi, W., Cao, J., Zhang, Q., et al.: Edge Computing: Vision and Challenges. IEEE Internet of Things Journal **3**(5), 637–646 (2016)
32. Tian, K., Ghobaei-Arani, M.: An autonomous energy-aware resource scheduling mechanism in serverless edge computing. Artificial Intelligence Review **59**, 96 (2026)
33. Vaño, R., Lacalle, I., Sowiński, P., et al.: Cloud-Native Workload Orchestration at the Edge: A Deployment Review and Future Directions. Sensors **23** (2023)
34. Wang, Z., Goudarzi, M., Aryal, J., Buyya, R.: Container Orchestration in Edge and Fog Computing Environments for Real-Time IoT Applications (2022), <https://arxiv.org/abs/2203.05161>
35. wasmCloud Project: wasmCloud: A Distributed Application Platform for WebAssembly. <https://wasmcloud.com/> (2019), accessed: 2026-02-16
36. WasmEdge Project: WasmEdge: Lightweight, high-performance WebAssembly runtime. <https://wasmedge.org> (2025), accessed: 2025-08-14
37. Yakubov, D., Hästbacka, D.: Comparative analysis of lightweight kubernetes distributions for edge computing: Performance and resource efficiency. In: Service-Oriented and Cloud Computing. pp. 81–95. Springer Nature Switzerland (2025)
38. Zhang, Y., Liu, M., Wang, H., et al.: Research on WebAssembly Runtimes: A Survey. ACM Trans. Softw. Eng. Methodol. **34** (2025)
39. Čilić, I., Krivić, P., Podnar Žarko, I., Kušek, M.: Performance Evaluation of Container Orchestration Tools in Edge Computing Environments. Sensors **23** (2023)