applied sciences

MDPI

Article

The Trade-Offs of Intelligent Orchestration in Kubernetes: A Multi-Infrastructure Case Study with the CODECO Toolkit

Georgios Koukis [1,*], Ioannis Dermentzis [1], Vassilis Tsaoussidis [1,*], Jan Lenke [2], Fabian Wölk [2], Daniel Uceda [3], Guillermo Sánchez [3], Miguel Angel Puentes Vallejo [4], Javier Serrano [4], Panagiotis Karamolegkos [5] and Rute C. Sofia [6]

1 Department of Electrical and Computer Engineering, Democritus University of Thrace, 67100 Xanthi, Greece; iderment@ee.duth.gr
2 Georg-August-Universität Göttingen, 37073 Göttingen, Germany; jan.lenke@cs.uni-goettingen.de (J.L.); fabian.woelk@cs.uni-goettingen.de (F.W.)
3 CDO Unit, Telefónica Innovación Digital, Ronda de la Comunicación s/n, 28050 Madrid, Spain; d.lozano.u@gmail.com (D.U.); guillermo.sanchezillan@telefonica.com (G.S.)
4 Informatic Systems Department, Universidad Politécnica de Madrid, 28031 Madrid, Spain; ma.puentes@upm.es (M.A.P.V.); javier.serrano@upm.es (J.S.)
5 Department of Digital Systems, University of Piraeus Research Center, 18534 Piraeus, Greece; pkaram@unipi.gr
6 Fortiss GmbH, 80805 Munich, Germany; sofia@fortiss.org
* Correspondence: gkoukis@ee.duth.gr (G.K.); vtsaousi@ee.duth.gr (V.T.)

Abstract

IoT monitoring systems, from traffic and pedestrian sensing in smart cities to building energy metering and industrial robot telemetry, increasingly run as containerised microservices on Kubernetes (K8s) clusters that span constrained edge nodes and cloud servers. Cognitive orchestration layers extend K8s with network-, data- and energy-aware placement, running on the same nodes as the monitoring workloads they manage, with their own footprint rarely reported. This paper proposes a component-level cost-attribution methodology and an automation-footprint trade-off analysis that relates the manual interventions removed to deployment latency, CPU, memory and estimated power consumed by each of its functional roles. We apply it to Cognitive Decentralised Edge–Cloud Orchestration (CODECO), an open-source cognitive orchestration framework, across six infrastructures, including three operational IoT deployments, ARM and x86 hardware, and vanilla K8s, K3s and KinD. The automated CODECO and CODEF workflow reduces the number of manual interventions from 72 to 9 (87.5%) under our action-count definition, with a total of 4.57 GB memory overhead, the most significant observed trade-off. Decomposing these costs by architectural layer reveals that within the tested cluster sizes, the installation time of the scheduling component shows no measurable growth and that of the learning component grows slowly, while cross-layer network management contributes the largest share of installation time and idle CPU consumption. Results show that orchestration costs arise in different architectural layers, indicating they can be placed and optimised independently without sacrificing automation benefits.

Keywords: cloud–edge–IoT continuum; Kubernetes; containerisation; AI/ML; heterogeneous systems; intelligent orchestration; trade-offs

Check for updates

Academic Editor: Dan Sui

Received: 29 September 2026
Revised: 29 October 2026
Accepted: 31 October 2026
Published: 5 November 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

IoT-enabled monitoring has become a core building block of critical infrastructure, transportation, energy and industrial systems, where networks of sensors, cameras and

embedded devices feed continuous data collection, analytics and decision-making [1] in smart cities [2], intelligent transportation systems [3], energy [4] and industrial production [5]. To meet latency, bandwidth and privacy requirements, the processing stages of these monitoring pipelines are increasingly placed close to the data sources, on edge gateways and single-board computers, while aggregation and long-term analytics remain in the cloud. Across this Cloud–Edge–IoT (CEI) continuum, the pipelines are typically packaged as containerised microservices and operated with Kubernetes (K8s) or one of its lightweight distributions designed for resource-constrained hardware [6] and evaluated on the edge devices that IoT workloads run on [7].

Real deployments mix hardware architectures (ARM/x86), resource profiles and network conditions, with workloads changing as sensors are added, links degrade or demand shifts [8,9]. A default scheduler places each pod once, based on node-level resource availability, unaware of network paths, data locality or energy. To close this gap, additional capabilities are introduced into the control plane, including scheduler extensions that improve placement decisions [10,11], platforms that push the control plane to edge nodes [12,13] or across multi-cluster boundaries [14], as well as cognitive orchestration layers that add context-aware, cross-layer and ML/AI-based decision-making [15–17].

However, a cognitive orchestration layer might be expensive. It may install controllers, learning agents, message brokers, databases and per-node network state on the same nodes that host monitoring workloads and applications. Evaluations of such layers report what they achieve (placement quality, QoS, energy savings) far more often than what they cost, rarely identifying where within the architecture that cost arises. Studies of lightweight K8s distributions quantify the footprint of the orchestrator itself [6,7,18], but not the footprint of the intelligence added on top of it. As a result, operators planning an IoT deployment lack the evidence needed to decide whether a given node can afford a cognitive orchestration layer, and which of its capabilities to enable. We address three research questions (RQs):

- RQ1: How many manual interventions does an automated, intent-driven orchestration toolchain save relative to the equivalent manual procedure?
- RQ2: What does the orchestration layer cost in deployment and lifecycle latency, CPU, memory and power, at idle and under sustained load?
- RQ3: Which functional role within the layer (configuration, scheduling, learning, network enforcement or data observability) incurs that cost, and how does it scale with cluster size and hardware?

We answer them with a component-level cost-attribution methodology and apply it to the open-source Cognitive Decentralised Edge–Cloud Orchestration (CODECO) [19] framework. CODECO itself is not a contribution of this paper, but serves as a representative case study since it implements all five functional roles as deployable components. Complementing it, the CODECO Experimentation Framework (CODEF) [20] automates reproducible experimentation via parameterised cluster and application specifications. The quantified trade-off compares the automation provided by a cognitive orchestration layer, measured by manual interventions removed, with the runtime footprint it imposes. The evaluation spans six infrastructures, including three operational IoT monitoring deployments, a media delivery testbed, a cloud environment and a local environment, with ARM and x86 hardware and three distributions (vanilla K8s, K3s and KinD). In summary, the main contributions are:

- A cost-attribution model and evaluation procedure for cognitive orchestration layers, which maps framework components to five functional roles, and separates per-cluster from per-node overhead. It quantifies resource consumption over time across key performance indicators (KPIs), including installation time, idle and loaded CPU/memory, estimated power and operator actions. It is framework-agnostic, requiring that the evaluated components can be enumerated and mapped to functional roles.

- A multi-infrastructure evaluation study across six environments, ARM and x86 hardware and three K8s distributions, using workloads of increasing inter-service coupling (isolated pods, a two-service application with a declared network channel and a multi-service benchmark) and three sustained request rates.
- Empirical cost attribution findings revealing that dominant components vary by resource, with network enforcement driving installation time and idle CPU while data observability dominates memory, and characterising how both scale with cluster size and workload density.

The remainder of this paper is organised as follows. Section 2 discusses related works and initiatives on edge–cloud orchestration and automation frameworks, while Section 3 presents CODECO, its architectural components, application model and use cases, as well as the experimentation framework CODEF. Section 4 details the experimental setup and methodology, while Section 5 analyses the obtained results. Finally, Section 6 discusses the main observations and our future directions, while Section 7 concludes the paper.

2. Related Work

The orchestration literature addresses two distinct questions. First, how placement and scaling decisions are made, and second, what the costs of system operation are. The first refers to scheduling theory, reinforcement learning and platform engineering, while the latter, including the deployment effort, resource consumption and architectural distribution of cost, is reported far less often. We discuss indicative literature works along these lines and Table 1 summarises these works.

Table 1. Representative edge–cloud orchestration literature. ✓ = addressed and quantitatively reported; ~ = partially addressed or reported qualitatively; × = not addressed.

Work/Class	Primary Focus	Evaluation Environment	Baseline	Manual Intervent.	Energy	Compon. Attrib.	HW Heterog.	Sustained Load
Scheduling and placement in Kubernetes								
Network-aware scheduling [21]	ILP placement over latency and bandwidth for fog	K8s fog testbed	Default scheduler	×	×	×	×	✓
Topology-aware scheduling [22]	Node/rack hierarchy for AI workload placement	K8s cluster, distributed training	Existing K8s schedulers	×	×	×	~	✓
Batch and gang scheduling [10]	Gang scheduling and queueing for batch and AI workloads	Production clusters	Default scheduler	×	×	×	×	✓
QoS-aware co-location [11]	Colocating latency-sensitive and best-effort workloads	Production clusters	Default scheduler	×	×	×	×	✓
ML-based K8s scheduling [15,23]	Forecast-driven scaling and placement	Single testbed	Default scheduler	×	×	×	~	✓
AI and ML-driven orchestration decisions								
RL-based offloading [24]	TD3 control with traffic-aware slicing	Simulation	Heuristic policies	×	~	×	×	✓
Graph-attentive MARL [25]	SLA-aware offloading and orchestration	Simulation	RL baselines	×	~	×	×	✓
Continuum coordination [26,27]	Online distillation; DVFS with offloading	Testbed	Static placement	×	~	×	~	✓
Multi-agent management [16,17]	Hierarchical closed-loop control	Simulation/testbed	Centralised control	×	~	×	~	✓

Table 1. *Cont.*

Work/Class	Primary Focus	Evaluation Environment	Baseline	Manual Intervent.	Energy	Compon. Attrib.	HW Heterog.	Sustained Load
Platforms and EU initiatives								
Multi-cluster [14]	Cluster peering, virtual nodes, cross-cluster networking	Testbed	Vanilla K8s, alternatives	×	×	×	~	✓
ENACT, COGNIT, EDGELESS, NEPHELE, MLSysOps [28–32]	Platform capability and use-case validation	Project testbeds	Varies/none	~	~	×	✓	~
CODECO-oriented studies [33,34]	Energy-aware placement; robotic edge orchestration	ARM far Edge; KinD	Vanilla K8s	×	✓	×	~	✓
Cost and footprint measurement								
Lightweight K8s distributions [6,7,18]	Idle footprint, control-plane performance (K3s, k0s, MK8s, KubeEdge)	Cloud VMs; edge devices	Vanilla K8s/each other	~	×	×	~	✓
CNI plugins [35]	Per-node overhead and performance of network plugins	K8s testbed	Each other	×	×	×	×	✓
Service-mesh sidecars [36]	Latency, CPU overhead per-pod proxies	Testbed	No mesh	×	×	✓	×	✓
Monitoring tools [37]	Energy, performance cost on containers	Container hosts	No monitoring	×	✓	~	×	✓
Security and monitoring add-ons [38]	Resource and energy cost of runtime security and network-monitoring	K8s cluster	No add-on	×	✓	✓	×	✓
Container energy observability [39,40]	Container- and pod-level power attribution	K8s clusters	Node-level power	×	✓	✓	~	✓
Edge–cloud traffic monitoring [41]	Power of orchestrated analytics on heterogeneous nodes	FPGA/ARM edge, GPU server; simulation	Cloud execution	×	✓	×	✓	✓
This work	Cost of cognitive orchestration and its architectural distribution	Six infrastructures, three K8s distributions	Vanilla K8s, manual procedure	✓	~	✓	✓	✓

2.1. Scheduling and Placement in Kubernetes

The default Kubernetes scheduler assigns pods through a two-stage filter-and-score procedure over node-level resource availability, evaluating each pod independently at admission time without revisiting the placement. Network-aware placement lets inter-node latency and bandwidth inform scoring [21], topology-aware placement incorporates node hierarchy into node selection for distributed AI training workloads [22], and broader surveys catalogue energy-, topology- and mobility-aware extensions proposed for edge settings [8,9]. Production-oriented schedulers address additional gaps, including gang scheduling for batch and AI workloads [10] and QoS-aware co-location of latency-sensitive and best-effort workloads [11]. Unlike existing methods that optimise placement within Kubernetes while ignoring extension overhead, cognitive orchestration introduces multi-layer components across network, data and decision planes with a non-negligible footprint.

2.2. AI and ML-Driven Orchestration Decisions

Learning-based methods have been applied to resource management and workload scaling across the edge–cloud continuum, targeting execution latency, resource utilisation and energy efficiency [42,43]. Within Kubernetes, forecasting models adapt horizontal pod autoscaling to fluctuating request patterns [15], and ML has been applied to container placement and resource allocation [23]. Reinforcement learning (RL) addresses the offloading and orchestration decision problem directly [24,25], while hierarchical and multi-agent architectures coordinate QoS-aware deployment [16,17] and adaptive techniques target the far edge [26,27]. These approaches replace static, operator-driven decisions with learned policies and several report substantial QoS improvements, but they are evaluated on the quality of the decisions rather than on the resources the decision layer consumes.

2.3. Platform and Framework Initiatives

Several EU-funded initiatives address AI/ML-driven orchestration across the continuum from a platform perspective, covering adaptive resource management [28], serverless execution and offloading [29,30], multi-cluster and far-edge coordination [31] and hierarchical multi-agent management [32]. Their published evaluations concentrate on capability and use-case validation rather than on the resources each platform consumes, so the footprint of the layer itself remains unreported. Prior work on the framework studied here has concentrated on what cognitive orchestration achieves. Sustainability-driven placement has been evaluated on ARM-based far edge testbeds, where it reduces cluster energy consumption relative to vanilla K8s under load and under injected fault conditions, and a case study on autonomous mobile robots reports reduced CPU consumption and more stable communication under mobility constraints of industrial monitoring deployments [33].

2.4. Cost of Orchestration and Intelligence

The cost of individual building blocks is mostly measured in isolation. Lightweight distributions such as K3s, k0s and MicroK8s have been compared for idle footprint and control-plane throughput under stress [6], with further comparisons covering life-cycle operations [18], serverless workloads [44] and KubeEdge, K3s and ioFog on real edge networks [7]. Comparable measurements exist for container network plugins, which differ substantially in the per-node overhead they add [35], for service-mesh sidecars, whose per-pod proxies can cost more than the application itself [36] and for monitoring tools, which carry a measurable energy and performance cost on container hosts [37]. These studies establish the baseline for a cognitive layer to be installed, but do not measure the footprint of placement intelligence, application-intent networking or data observability.

More recent studies extend operational measurements to energy. Smart-city traffic edge deployments show energy consumption several times lower than cloud execution across heterogeneous nodes [41]. Our prior work quantified the continuous per-node resource and energy overhead of Kubernetes security and network monitoring tools [38]. Similarly, carbon-aware container scheduling surveys evaluate savings achieved rather than the consumption of the decision layer itself [45]. This measurement is further constrained by instrumentation, relying on hardware counters or power models that limit direct measurement on ARM and virtualised nodes [39,40,46] (cf. Section 4.2.3).

3. Case Study and Functional Decomposition: The CODECO Framework

CODECO [19] is an open-source orchestration framework that extends Kubernetes native capabilities with cognitive and policy-driven control for distributed edge–cloud orchestration of microservices. It interoperates with standard Kubernetes but extends it with cross-layer abstractions (compute, network, data) and ML-/rule-based decisions to

deliver an orchestration plane that continuously adapts to application requirements, end-user intent and infrastructure conditions, ranging from data characteristics and resource footprint to network dynamics and user-level objectives, without requiring extensive manual customisation. CODECO consists of five collaborative components that constitute the complete framework, as illustrated in Figure 1, while Table 2 summarises its decomposition, which is the basis of the cost attribution in Section 5. For each component, we summarise the mechanism it implements and how it is deployed, i.e., once per cluster or on every node, since this deployment structure determines how its cost scales.

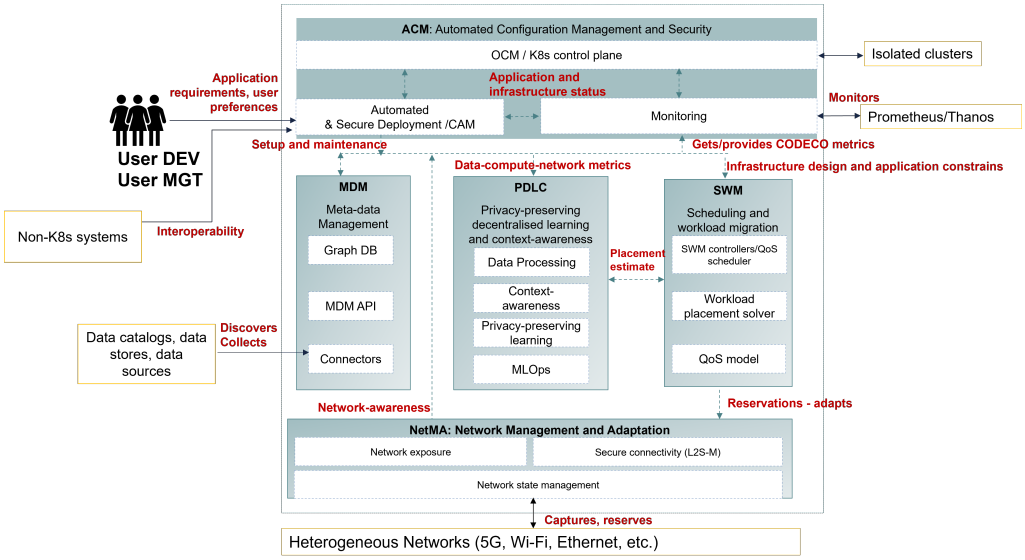


Figure 1. CODECO high-level architecture.

Table 2. Architectural breakdown of a cognitive orchestration layer in the case study framework.

Functional Role	Component	Functionality and Deployment	Closest K8s Counterpart
Configuration and automation	ACM	Operator on the control plane; translates the application model into deployable specifications; installs the other components, a message broker and a graph database; integrates the Prometheus stack	Operators, Helm-based installers
Scheduling	SWM	Replaces the default scheduler; solves placement as a constrained optimisation over QoS, compute, network and energy metrics; (re)places workloads on recommendation	Scheduler plugins, Volcano, Koordinator
Learning and context	PDLC	Agents on control-plane and worker nodes; GNN-based modelling of cluster state and RL-based recommendations expressed as CRDs; learning is performed locally	Learning-based controllers and autoscalers
Network enforcement	NetMA	Exposes latency, bandwidth and link state; instantiates declared service channels as programmable per-node paths via SDN integration (L2S-M)	CNI plugins, SDN controllers, service meshes
Data observability	MDM	Captures and correlates application, system and network metadata in a graph store, backed by Kafka, Zookeeper and Neo4j	Observability and metadata stacks

3.1. Automated Configuration Management (ACM)

ACM is the user entry point of CODECO. It translates application requirements over compute, network and data into deployable specifications and exposes the interface through which users interact with the framework. ACM builds on Open Cluster Management [47] for control-plane visibility across clusters and is itself installed as a Kubernetes meta-operator, running on the control plane, so its cost is incurred once per cluster. Application requirements are submitted as a `CodecoApp` custom resource, i.e., the CAM of Section 3.6, which ACM propagates to the other components through their own custom resources (CRs). ACM is also the installer of the remaining four components and of their supporting infrastructure, including a message broker and a graph database. The control plane of the layer is therefore a set of co-installed distributed systems, each with its own footprint, and it is itself observed through the Prometheus stack alongside the workloads it manages.

3.2. Scheduling and Workload Migration (SWM)

SWM replaces the default Kubernetes scheduler and formulates placement as a constrained optimisation over QoS, compute, network and energy metrics, solved to match the intents declared in the application model. It comprises two subcomponents running on the control plane. The `QoS Scheduler` is built on the Kubernetes scheduling framework as an additional scheduler (`qos-scheduler`). While the default scheduler places each pod individually, the QoS Scheduler places all pods of an application group jointly, treating placement as a graph problem over workloads and channels between them. The `Workload Placement Solver` is a gRPC service receiving a QoS model with an application model (workloads and the channels between them, with their bandwidth and delay requirements) and an infrastructure model, compiled from node, network-node, link and path resources. Placement is solved as an integer linear program with $x_{w,n} \in \{0,1\}$ indicating that workload w is placed on node n and $y_{c,p} \in \{0,1\}$ that channel $c = (w, w')$ is mapped to network path p, subject to single-node placement ($\sum_n x_{w,n} = 1$), node resource capacity ($\sum_w r_w^k \, x_{w,n} \leq R_n^k$), link bandwidth constraints ($\sum_c \sum_{p \ni \ell} b_c \, y_{c,p} \leq B_\ell$), and path latency bounds ($\sum_p d_p \, y_{c,p} \leq D_c$). It returns the placement as an `AssignmentPlan` resource and revisits placements when PDLC recommends re-scheduling due to infrastructure conditions or performance changes.

3.3. Privacy-Preserving Decentralised Learning and Context-Awareness (PDLC)

PDLC supplies the context on which SWM's decisions depend, acting as a recommender so that new profiles or constraints require no retraining of the scheduler. Its context-awareness subcomponent aggregates metrics exposed by ACM, NetMA, and MDM into a cost per node for the performance profile declared in the CAM. For the greenness profile, with $n_e(i)$ denoting the energy consumed by node i across all processes and $L_e(i) = \sum_j l_e(i,j)$ its egress transmission energy, nodes are ranked either by a compute-centric cost, $g(i) = n_e(i)$, or by one accounting for transmission energy, $g(i) = n_e(i) \, L_e(i)$. The decentralised-learning subcomponent combines GNN forecasts of node metrics with RL to produce node recommendations emitted as Kubernetes CRs. Operating across both master and worker nodes, PDLC performs learning locally.

3.4. Network Management and Adaptation (NetMA)

NetMA exposes network attributes to PDLC and SWM, and instantiates the service channels declared in the application model as secure, programmable pod-to-pod paths through SDN integration (L2S-M). A per-node probe measures link- and path-level attributes, including bandwidth, latency, path length, failures, node degree and link energy, which feed the costs of PDLC, while a programmable link-layer switch on each node forms

a VXLAN overlay on which an SDN controller installs the flow rules of a requested path before reporting the channel to SWM. Since a channel with bandwidth and delay bounds must exist as a configured path hosting a participating service, NetMA maintains state on each such node, allowing link-layer connectivity to be created and reconfigured between pods independently of the default cluster network.

3.5. Metadata Manager (MDM)

MDM provides data observability by capturing and correlating application, system and network metadata in a graph representation. Connectors emit metadata as events, i.e., JSON documents carrying an operation type, connector identifier, timestamp, payload of entities and relationships, which are queued on a message backbone and transformed into a graph store queried by the other components through the MDM API. It supports data traceability and performance impact analysis, e.g., identifying constraints and how those bound placement, complementing the system and network monitoring from the other CODECO components. It is deployed once per cluster and backed by Kafka, Zookeeper and Neo4j, which account for most of its resource footprint.

3.6. The CODECO Application Model (CAM) and Capabilities

The CODECO Application Model (CAM) is a YAML-based, declarative user-configured manifest established during application deployment setup. It specifies the application's constituent services and runtime requirements, which ACM assembles and validates (cf. Table 3). The finalised CAM informs cluster sizing and resource reservations, before placement, migration, and end-to-end lifecycle management across CODECO components. Each energy, network and QoS field declared in the CAM is consumed by a specific CODECO component and produces an operational effect. The `performanceProfile` and `appEnergyLimit` fields feed SWM's ILP-based scheduler, which combines computational energy telemetry (via Kepler) with network-level metrics into greenness heuristics that guide pod placement, establishing the CAM's declarative surface.

Table 3. Indicative CAM fields (YAML keys) that extend vanilla Kubernetes and their intent.

Field/YAML Key	Description
`performanceProfile`	Optimisation goal (e.g., energy efficiency vs. performance/cost). Used to steer scheduling/placement decisions (e.g., prioritise greener/lower-energy nodes when set to `Greenness`).
`appEnergyLimit`	Application-level energy budget (provided as a string, e.g., ``20''). Enables energy-aware scheduling decisions based on an upper bound of energy consumption.
`appFailureTolerance`	Acceptable failure tolerance level for the application. Can be an empty string (``'') (i.e., no explicit tolerance provided).
`schedulerName`	Scheduler selection for the workload (e.g., `qos-scheduler` instead of K8s' `default-scheduler`).
`serviceChannels`	Declares network channels between microservices with QoS requirements/guarantees (i.e., application-intent networking beyond vanilla K8s).
`serviceClass`	Network traffic/service class (e.g., `ASSURED`, `BESTEFFORT`). Used to request bandwidth/latency guarantees.
`bandwidth`	Requested/guaranteed bandwidth between services (e.g., ``5M'' for 5 Mbps).
`maxDelay`	Maximum allowed end-to-end delay/latency between services (e.g., ``10ms'' as specified).
`complianceClass`	Compliance requirements for the application (e.g., data handling constraints, placement/processing).
`qosClass`	Service tier for prioritisation (e.g., `Gold` > `Silver` > `Bronze`).
`securityClass`	Application security requirements (e.g., security level expected from hosting environment/policies).

3.7. CODECO Experimentation Framework (CODEF)

CODEF [20] is an open-source framework developed to accelerate experimentation in containerised edge–cloud environments. Its microservice-oriented, modular design promotes flexibility, extensibility and the seamless incorporation of new technologies. CODEF provides abstractions that cover the full experimental workflow, i.e., from node allocation and cluster/application deployment to declarative experiment specification and end-to-end automation from system setup through results visualisation. It offers a comprehensive, adaptable approach to manage heterogeneous resources across diverse infrastructures. By building on and extending well-established tools (e.g., Ansible, kubeadm, Bash), CODEF ensures automation, interoperability and reproducibility for complex scenarios.

3.8. Use Cases and Infrastructures

The evaluation relies on four use-case infrastructures, three of which host operational IoT monitoring deployments. They determine the hardware, Kubernetes distribution and network conditions under which a cognitive orchestration layer would run in practice. Their full specifications and objectives are described in [19].

Smart-City Infrastructure Monitoring (UC1, Göttingen, Germany): Thermal cameras and LiDAR sensors attached to NVIDIA Jetson edge workers feed real-time traffic and pedestrian analytics, with a cloud-hosted control plane and continuous dashboards under privacy-preserving data handling. This deployment represents camera-driven monitoring on capable ARM edge nodes connected to a remote control plane.

Media Delivery Streaming (UC2, Telefónica, Spain): A Media Delivery System spans a virtualised control plane and Raspberry Pi 5 edge workers, orchestrating both compute placement and network paths. By integrating the system with CODECO, the network management component (NetMA) exposes ALTO-style [48] topology and cost maps, enabling per-request selection of the optimal route.

Smart Energy Grids (UC3, UPM, Spain): Energy-related IoT data from university campus buildings are processed near the source, while aggregation, forecasting, and optimisation run in the cloud under strict latency and reliability constraints. Through cluster-level coordination, UC3 enhances energy efficiency, sustainability, and resilience, with evaluation focusing on dynamic energy-cluster formation. Overall, this deployment represents sensor-driven monitoring across heterogeneous x86 hardware.

Industrial Robot Telemetry and Control (UC4, Fortiss, Germany): A fleet of Wi-Fi-connected autonomous mobile robots (AMRs) serves as K3s worker nodes hosting containerised telemetry and control services, while CODECO manages deployment, placement, and migration across the compute and network layers to mitigate wireless interference and intermittent links. Representing mobile, resource-constrained IIoT nodes, this deployment targets lower task latency, improved energy and network efficiency, and robust operation.

4. Experimentation Setup, Methodology and Metrics

This section outlines the experimental approach adopted in this work (cf. Figure 2), covering the methodology, use-case technical specifications, KPIs and associated metrics, and evaluated workload scenarios. All experiments are fully reproducible, with the complete implementation, configuration scripts, and collected results openly available.

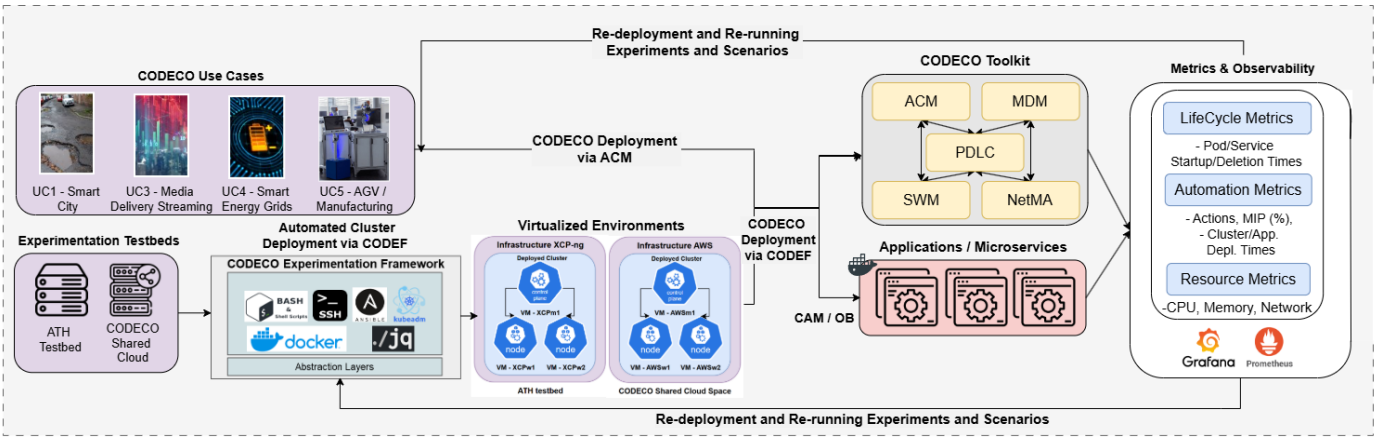


Figure 2. Experimentation methodology workflow.

4.1. Methodology

Table 4 lists the seven experiments executed together with the RQs they answer, scope and utilised metrics. Every runtime experiment compares CODECO against the same Kubernetes environment without CODECO, on the same infrastructure (vanilla K8s, K3s or KinD). Since CODECO operates on top of Kubernetes and introduces advanced cognitive capabilities without a native counterpart, the experiments differ in what they compare:

- Automation tooling (E1, phases I–II): manual provisioning and Kubernetes installation against CODEF. These phases are orchestrator-agnostic, since Kubernetes deployments can automate them with infrastructure-as-code tooling.
- Installer automation (E1, phase III): manual installation of the CODECO components against their automated installation by ACM. Since these components have no vanilla counterpart, this baseline is constructed from their documented installation procedure.
- Installation cost (E2): the absolute installation time of CODECO, decomposed per stage and per component, the basis for the per-role attribution.
- Complete platform (E3, E5–E7): the full runtime CODECO path (ACM translation, SWM placement with PDLC context, NetMA and MDM) against the default Kubernetes. In E3 pods declare no service channels, covering only admission and placement.
- Complete platform with application-intent networking (E4): as E3, E5–E7, but NetMA additionally provisions a service channel with declared bandwidth and delay bounds, which the baseline does not create.

Table 4. Summary of runtime experiments (CODECO against Kubernetes).

Exp.	RQ	Workload/Procedure	Scope of Comparison	Metrics
E1	RQ1	Deployment procedure, phases I–III	Phases I–II: manual vs. CODEF; phase III: manual vs. ACM installer	K_a, C_a, MIP
E2	RQ2, RQ3	Toolkit installation	Absolute cost per stage and component	T_{inst}
E3	RQ2	App. 1: pause pods, 10–150 per batch	Complete runtime path	T_{ready}, T_{del}
E4	RQ2, RQ3	App. 2: two services, 1–50 replicas	Runtime path plus a service channel absent in the baseline	T_{ready}, T_{del}
E5	RQ2	App. 3: Bookinfo	Complete runtime path	T_{ready}, T_{del}
E6	RQ2, RQ3	Idle cluster	Installed platform	CPU, memory, $\hat{P}$
E7	RQ2, RQ3	Bookinfo at 25, 50, 100 req/s	Installed platform under load	CPU, memory, req. latency

The experimentation methodology follows IETF BMWG guidelines for benchmarking virtualised and containerised infrastructure [49,50], together with established reporting principles for cloud performance evaluation [51]. While these prescribe how such overheads should be measured, they do not specify how the overhead should be attributed to its functional roles. We (i) report the full hardware and software configuration of each system under test, (ii) apply identical measurement boundaries to both configurations, (iii) repeat every configuration and report variability, and (iv) release the specifications, scripts and raw measurements to repeat the experiments. Each scenario is executed ten times on a freshly deployed cluster, after a discarded warm-up run and inter-iteration pauses. Where applicable, we report *MIN*, *MAX*, *AVG* and *SD*, the mean difference Δx, and the β (s/node) with its 95% confidence interval (CI), $\Delta x \pm t_{0.975,\nu}\sqrt{s_C^2/n_C + s_K^2/n_K}$, with ν given by the Welch–Satterthwaite approximation, to show variability across experiment stages. We capture deployments with timestamps at key phases (e.g., Idle, Ready, Delete) and capture resource usage via cluster telemetry, utilising Prometheus for metrics scraping, complemented by script logging. Our experiments evaluate CODECO against vanilla K8s, which serves as our baseline. Across the evaluated hardware-heterogeneous edge–cloud UCs, scenarios and applications, we measure three metric classes targeting defined KPIs:

- Lifecycle: measure the startup and deletion times of pods and services.
- Automation: quantify the required actions and the Manual Intervention Percentage.
- Resource-footprint: capture CPU, memory and power during cluster operations.

Therefore, applying the framework to another orchestration layer requires (i) assigning its components to the five functional roles of Table 2, (ii) recording for each whether it is deployed per cluster or per node, and (iii) running E2–E7 with the layer's own deployment path, with E1 being procedure-specific, depending on its documented installation steps.

4.2. Key Performance Indicators, Metrics and Cost Model

To quantify the performance of the CODECO toolkit and the efficiency of automated multi-service deployment within single-cluster edge–cloud environments, this work focuses on a set of KPIs and corresponding experimentation metrics:

- KPI1 (RQ2, RQ3): Time taken to set up and configure multiple clusters and services based on scalable single clusters.
- KPI2 (RQ1): Percentage of manual intervention required against the documented manual procedure when setting up clusters and services.
- KPI3 (RQ2, RQ3): Overall resource footprint (CPU, RAM and estimated power overhead) in idle and loaded cluster state.

For a measured quantity x such as installation time, readiness time, CPU, memory, or estimated power, let x_K and x_C denote its values under the baseline and under CODECO on the same infrastructure, respectively. We quantify performance differences using the absolute overhead ($\Delta x = x_C - x_K$), relative overhead ($\rho_x = \Delta x / x_K$), and ratio ($R_x = x_C / x_K$). Furthermore, we decompose this overhead across the functional roles $r \in \mathcal{R} = \{\text{ACM}, \text{SWM}, \text{PDLC}, \text{NetMA}, \text{MDM}\}$, incorporating a residual term ε to account for shared services (e.g., monitoring) and unattributed effects, expressed as $\Delta x = \sum_{r \in \mathcal{R}} x_r + \varepsilon$. Each framework component is deployed either once per cluster (as `Deployment` or `StatefulSet`) or once per node (as `DaemonSet`). For a cluster with n nodes, we model its resource cost as $x_r(n) = \alpha_r + \beta_r n$, where α_r represents the fixed per-cluster baseline overhead and β_r denotes the marginal cost per added node. Parameters α_r and β_r are estimated via ordinary least squares (OLS), with β_r reported alongside its 95% confidence interval.

4.2.1. Percentage of Manual Intervention (E1) and Automation

End-to-end deployment in edge–cloud environments involves a sequence of high-level manual actions across multiple stages from scratch, including:

1. VM and OS provisioning (Phase I), including resource allocation, OS bootstrapping, SSH access setup, host configuration and base package installation.
2. Kubernetes installation (Phase II), covering container runtime, Kubernetes setup and networking configuration (CNI installation).
3. Service deployment (Phase III), requiring preparation of dependencies (e.g., Helm, yq, cert-manager) and deploying the associated software components.

We define a manual action as a single operator intervention required by the documented deployment procedure, i.e., a command the operator must issue, or a configuration file, manifest or specification the operator must author or edit. A scripted sequence that runs without further operator input counts as one action regardless of its internal steps. This definition accounts for the interventions required, with the baseline defined as the standard manual procedure where each command is issued individually.We use the Manual Intervention Percentage (MIP), computed by identifying all major deployment and installation phases and quantifying the proportion of user-defined manual actions required in the CODECO/CODEF environment (C_a) relative to the documented manual procedure (K_a), as defined in Equation (1), where C_a denotes the number of manual actions required in the CODECO/CODEF environment and K_a the corresponding number of manual actions in the documented manual procedure.

$$\text{MIP} = \frac{C_a}{K_a} \times 100\% \tag{1}$$

Therefore, a lower MIP indicates a higher degree of automation. For the cluster deployment and Kubernetes installation (phases I and II), CODEF-based automation (Ansible/Bash) is employed, providing end-to-end provisioning logic from host preparation and cluster bootstrap to application deployment across heterogeneous environments. In contrast, service installation (phase III) relies on the CODECO *post_deploy.sh* script, which installs the complete CODECO toolkit via ACM. For phases I and II, baseline actions are derived directly from CODEF's corresponding Ansible playbook task definitions, while automated actions emerge from the user-specified configuration files that those tasks automate. The baseline of phase III includes the manual installation of the same CODECO components, which have no vanilla K8s counterpart.

Complementary to the MIP, CODEF reduces deployment to a reusable, parameterised workflow that requires a one-time configuration, making setup costs negligible relative to the automated execution runs it enables. While manual deployment requires a chain of per-node configurations (OS provisioning, network setup, container runtime installation and application deployment), CODEF reduces this to populating two configuration files, creating an Ansible playbook for unsupported applications and triggering execution.

4.2.2. Installation and Lifecycle Latency (E2–E5)

This metric captures the time required for cluster deployment across its distinct stages. For deployments performed with CODEF, we explicitly measure node discovery (ND), operating system installation (OS_I) and Kubernetes installation ($K8S_I$), as CODEF manages the full node lifecycle prior to cluster formation. In contrast, emulation environments based on KinD [52] assume pre-configured container-based nodes and therefore include only the cluster deployment and application installation stages. Each stage is measured based on the completion events to capture deployment overhead across different cluster sizes.

To assess pod initialisation overhead (E3), both configurations share identical measurement boundaries. The start event t_0 is recorded immediately before submitting the manifest via `kubectl apply` (a Deployment for the baseline, a `CodecoApp` for CODECO), while the end event marks when the final pod reports the `Ready` condition. This interval includes admission, scheduling, container creation, network attachment and readiness reporting in both cases. Under CODECO, it incorporates ACM application model translation, SWM placement with PDLC context and NetMA network path establishment. Timestamps are captured via a watch-based client.

Evaluating service instantiation overhead (E4, E5) relies on the same initial start event while extending the completion criteria, marking a service as ready only when all its endpoints are fully provisioned and an in-cluster probe receives its first successful response. The deletion interval spans from a client-side timestamp recorded immediately prior to issuing `kubectl delete` until the API server no longer returns any of the targeted resources, including pods, deployments and endpoints. This cluster lifecycle metric captures CODECO's end-to-end service management from ACM/SWM-driven instantiation and placement based on PDLC recommendations, to endpoint availability through NetMA channel establishment and conversely, from deletion request to complete teardown.

4.2.3. Resource Consumption (E6, E7)

This metric captures the overall resource efficiency and estimated power footprint of CODECO and deployed applications. Specifically, we measure CPU utilisation (in %), memory consumption (in GB) and power usage (in W). CPU and memory are reported at two levels: node-level, available for the cluster footprint, and container working-set memory for the per-component attribution. Resource metrics are collected on control plane and worker nodes, reporting both per-node CPU and memory, but also cluster- and system-level power. CPU utilisation is measured individually for each node under idle conditions, both with and without CODECO. For the three-node INF6 cluster, we additionally report the sum of the per-node CPU utilisation percentages as an aggregate descriptive metric. The overhead is calculated as the difference in percentage points (pp) and relative changes in %, between the two configurations under otherwise identical conditions. The data are collected via Prometheus exporters or *top* commands (for system-level metrics), comparing CODECO with vanilla K8s or KinD, highlighting how CODECO's scheduling, context- and network-awareness impact resource utilisation. The monitoring stack (Prometheus, node exporters, Kepler) runs in both configurations, so it is part of the baseline and not of the reported overhead. To determine the cluster's cost under load (E7), we repeat the footprint measurements while driving the Bookinfo application at different request rates (25, 50 and 100 requests per second), each sustained for ten minutes after an established idle baseline. Load was generated using k6, which issues requests at a fixed rate. Requests target the Bookinfo endpoint through a NodePort service. Request-driven load serves as a proxy for the query traffic that dashboards and API clients generate against monitoring services.

Estimating power draw in environments which combine VM-based infrastructures (e.g., CODEF) and containerised local clusters (e.g., KinD) poses significant challenges. Tools such as Kepler [39] attribute energy at the container and process level by reading hardware power interfaces, particularly RAPL (Running Average Power Limit) [53], which exposes CPU package and DRAM energy counters on x86 platforms, distributing the measured node power across workloads in proportion to their resource utilisation. This approach is constrained in some evaluated environments here, as RAPL is unavailable on ARM-based platforms (Raspberry Pi clusters) and in virtualised environments the underlying counters are not exposed to the guest [46]. Thus, Kepler falls back to pre-trained regression models whose accuracy depends on how closely the training platform matches the target, since

such models are system-specific and vary with CPU model and architecture. Given these constraints, we adopt an established linear power model [54]:

$$\hat{P}_i = P_{\text{idle}} + (P_{\text{max}} - P_{\text{idle}})\, U_i + k_{\text{mem}} M_i, \tag{2}$$

where $\hat{P}_i$ is the estimated power draw of node i (in W), P_{idle} and P_{max} its idle and peak power, $U_i \in [0, 1]$ its measured CPU utilisation, M_i its memory in GB, and $k_{\text{mem}} = 0.2$ W/GB the memory power coefficient. For the INF6 we use $P_{\text{idle}} = 35$ W and $P_{\text{max}} = 95$ W, obtained by allocating host-level values by the VM's share of host threads, $P_{\cdot,\text{VM}} = P_{\cdot,\text{host}}\, v_{\text{VM}}/v_{\text{host}}$; for INF1 host we use $P_{\text{idle}} = 60$ W and $P_{\text{max}} = 165$ W. Summing Equation (2) across all nodes gives the total power overhead:

$$\Delta\hat{P} = (P_{\text{max}} - P_{\text{idle}}) \sum_i \Delta U_i + k_{\text{mem}} \sum_i \Delta M_i. \tag{3}$$

While $\Delta\hat{P}$ is independent of P_{idle}, the relative overhead $\rho_{\hat{P}}$ is controlled by P_{idle} through its denominator. We therefore report $\Delta\hat{P}$ (in Watts) as our primary metric alongside a sensitivity range reflecting: (i) $P_{\text{max}} - P_{\text{idle}}$ varied by ±33%, and (ii) k_{mem} swept from 0.2 W/GB to 0.392 W/GB (the Cloud Carbon Footprint benchmark [55]).

4.3. Evaluation Use Cases, Infrastructures and System Specifications

This section describes the underlying infrastructures of the CODECO technical use cases. We evaluate the deployed scenarios across six heterogeneous environments (INF1 to INF6), corresponding to the four UCs presented, a local deployment with KinD and an AWS-based experimentation environment. These include two experimental and four on-premises real-world testbeds, covering diverse hardware (ARM/x86, edge devices to servers), network conditions and Kubernetes flavours (vanilla K8s, K3s). In addition, to evaluate scalability-related KPIs, we also run a subset of experiments on a local KinD cluster hosted on a personal computer (in INF1). Detailed infrastructure specifications and deployment methods are summarised in Table 5.

4.4. Experimental Applications

This section defines the application workloads and tools used to evaluate CODECO's deployment and orchestration performance. We execute a range of applications, starting from simple batch pod deployments, extending to a dummy application and finally a realistic microservice benchmark suite (Bookinfo), to study initialisation, scaling and teardown behaviour under heterogeneous nodes and varying workload profiles. In addition to standard *kubectl*-based deployments, applications are also deployed through the CAM.

Application 1: Pod Readiness and Deletion per Batch Size. In our experiments, we launch batches of 10–150 pods on cluster workers using a *k8s.gcr.io/pause:3.1* minimal image, with the pods executing no application workload. For each batch we record the elapsed time until pods reach the *Ready* condition and report the batch-average readiness and deletion time. Since the image performs no computation and declares no inter-service dependencies, the measured latency isolates the orchestration path itself, including admission, scheduling, network attachment and readiness reporting, from any application-level startup cost.

Application 2: Dummy Frontend-Backend. We deploy a lightweight backend and a frontend service, packaged as independent containers and establishing a bidirectional service channel between them to simulate a typical microservice workflow. In particular, the backend provides a simple HTTP endpoint (e.g., */api/hello*) that returns greeting text, while the frontend issues requests to this endpoint and displays the responses. The distinction from Application 1 is the presence of a declared inter-service dependency which requires NetMA to establish an actual connectivity path between the two services, isolating the cost

of application-intent networking across increasing replica counts. This pattern mirrors monitoring pipelines in which a sensor-facing service streams data to an analytics service over a link with explicit bandwidth and delay requirements in smart-city monitoring.

Table 5. List of use case infrastructures and specification details.

Name	Domain	Infrastructure Specifications	Environment	Deploy
INF1	Local deployment	Intel Core i9-10900K CPU @ 3.70 GHz, 10 cores/20 threads (x86_64), 64 GB DDR4 RAM, NVIDIA GeForce GTX 1660 SUPER GPU, 1 TB SSD storage, running Ubuntu 22.04 with Linux kernel 6.8.0.	KinD	post_deploy
INF2	Smart-City Infrastructure Monitoring (UC1)	*Control plane:* OpenStack VM, 16 GB RAM, 148 GB storage, 4 cores (AMD EPYC, @ 3.245 GHz). *Workers:* NVIDIA Jetson AGX Orin, 64 GB RAM, 12 vCores (ARMv8), 520 GB storage.	Vanilla K8s (VM-based)	post_deploy
INF3	Media Delivery Streaming (UC2)	*Control plane:* Proxmox VM (16 GB RAM, 32 GB storage, 8 cores, QEMU). *Workers (2):* Raspberry Pi 5, Broadcom BCM2712 quad-core 64-bit Arm Cortex-A76 @ 2.4 GHz, 4 GB RAM, 32 GB, Raspberry Pi OS.	K3s	post_deploy
INF4	Smart Energy Grids (UC3)	*Control plane:* Intel Core i9-10900 @ 2.80 GHz, 10 cores/20 threads (KVM); 32 GB DDR4 DIMM @ 2933 MT/s; 1 TB NVMe SSD. *Worker 1:* Intel Core i7-7700K @ 4.20 GHz; 4 cores/8 threads (KVM); 16 GB DDR4 DIMM @ 2400 MT/s; 256 GB SATA SSD + 1 TB HDD. *Worker 2:* Intel Core i7-860 @ 2.80 GHz; 4 cores/8 threads (VT-x); 8 GB DDR3 DIMM @ 1333 MT/s; 2×500 GB HDD (SATA).	K3s	post_deploy
INF5	Industrial Robot Telemetry and Control (UC4)	*Control plane:* Intel(R) Core(TM) i7-8550U CPU @ 1.80 GHz; 16 GB DDR4. *Workers (7):* Raspberry Pi 4 (ARM64).	K3s-edge	post_deploy
INF6	Shared Cloud Space (AWS)	Per node specifications (3 nodes): Intel Xeon Platinum 8259CL @ 2.50 GHz, 8 physical cores, 16 threads (KVM); 64 GB DDR4 DIMM @ 3200 MT/s; 97 GB NVMe SSD.	Vanilla K8s (VM-based)	CODEF

Application 3: Bookinfo Suite. Bookinfo [56] is a multi-service HTTP benchmark application composed of interdependent microservices (*productpage*, *details*, *reviews*, *ratings*) with varying resource profiles that interact through HTTP, simulating controlled traffic and service behaviour. Deployed in full, its multi-component topology exercises placement, channel establishment, startup latency, and orchestration efficiency at realistic granularity, complementing the isolated evaluations of Applications 1 and 2.

5. Experimental Results

This section reports the experimental results across the three KPIs defined above: the reduction in manual interventions achieved through automation, the deployment and lifecycle latency incurred across cluster and application scenarios, and the resource and power footprint of the CODECO control plane relative to vanilla K8s.

5.1. Percentage of Manual Intervention

Table 6 summarises the manual interventions required to: (i) provision a cluster (VM and OS), (ii) install Kubernetes and (iii) deploy the CODECO toolkit, comparing

the manual approach and the automated CODECO-driven procedures, complemented by CODEF processes, specifically for the first two phases. The required actions for these phases are enumerated directly from CODEF's Ansible playbook tasks.

Table 6. Manual procedure per deployment phase against automated (CODECO + CODEF). Lower MIP values indicate a higher degree of automation.

Phase	Method	Auto Level	Actions	MIP (%)
Cluster Deployment	Manual CODEF	None Full	19 3	15.79
K8s Installation	Manual CODEF	None Full	42 4	9.52
Service Deployment	Manual CODECO	None Full	11 2	18.18
All Phases	Manual CODECO + CODEF	None Full	**72** **9**	**12.5**

Regarding cluster deployment (phase I), the baseline manual approach requires a total of 19 discrete actions, including VM instantiation on the target infrastructure, OS image selection, SSH key generation and distribution, firewall rule setup and more. CODEF reduces the user-defined actions to 3 (i.e., MIP = 15.79%), including the creation of the configuration file (the infrastructure manager used, user credentials and nodes' specifications), the declaration of the above file in the deployment script and the process execution. In terms of Kubernetes installation (phase II), actions such as Docker/container runtime installation, kubeadm initialisation, certificate generation, Kubernetes distribution and CNI plugin deployment, worker node joining and verification steps must be performed, a total of 42 actions. CODEF automates the procedure by requiring the user to select the Kubernetes distribution, CNI plugin, any other network parameter, and finally run the script, i.e., 4 actions (9.52% MIP). Focusing on service deployment (phase III), CODECO's components have no vanilla Kubernetes counterpart, therefore, quantifying the manual actions required to install equivalent components without ACM relies on following their documented installation procedures and measuring the automation provided by the installer. Under this scenario, the manual service deployment process includes the retrieval of cluster information and credentials, installation of prerequisites, repository cloning, validation and functionality verification (11 discrete actions). In contrast, CODECO automates this workflow through an installation and validation script, resulting in an MIP of 18.18%.

MIP depends on the granularity of the defined action, applied to both workflows. Across the full deployment lifecycle, the combined CODECO/CODEF workflow reduces operator interventions from 72 to 9 (12.5% MIP), corresponding to an 87.5% reduction. Of the 63 actions removed, 54 (85.7%) are removed by CODEF in phases I–II with infrastructure-as-code tools, while 9 (14.3%) by CODECO's installer in phase III. If a finer action definition inflates the baseline and automated counts by factors f_K and f_C, then MIP $= (f_C/f_K) \times 12.5\%$. The reduction therefore stays above 50% unless the automated procedure is counted more than four times as finely as the baseline, and vanishes only at a ratio of eight.

5.2. Cluster Deployment and Installation Times

A central question for any cognitive orchestration framework is whether its measured benefits derive from the cognitive decision layer itself, or from the deployment-automation tooling that precedes it. Figure 3 represents the mean deployment times as the number of nodes scales, obtained from (i) CODEF and (ii) a KinD-based environment.

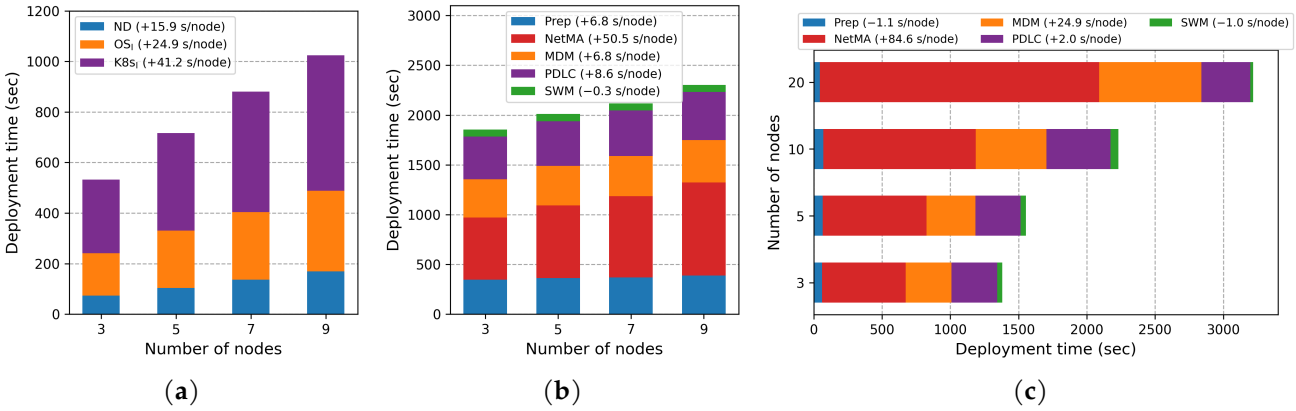


Figure 3. Mean deployment time per nodes' number: (**a**) CODEF stages (ND, OS and K8s installation), (**b**) CODECO installation via CODEF, and (**c**) CODECO installation via KinD.

Figure 3a breaks down CODEF deployment time into Node Discovery (ND), OS installation (OS_I) and K8s installation ($K8S_I$). Scaling from 3 to 9 nodes, end-to-end deployment grows from 532.3 s to 1023.7 s (+92.3%, $\beta = 82.0$ s/node [68.1, 95.8]). $K8S_I$ dominates (52–55%) and is the primary scaling driver ($\beta = 41.2$ [28.3, 54.1]), followed by OS_I (30–31%, $\beta = 24.9$ [19.4, 30.3]) and ND (14–16%, $\beta = 15.9$ [15.0, 16.9]), scaling in absolute terms from 291.2 to 535.0 s, 167.7 to 319.9 s and 73.4 to 168.8 s, respectively. Figure 3b shows CODECO installation times per component. For 3 nodes, installation requires 1855.9 s, increasing to 2011.1 s (5 nodes), 2117.7 s (7 nodes) and 2303.1 s for 9 nodes (+24.1% overall, $\beta = 72.4$ s/node [50.5, 94.3]). NetMA constitutes the largest and fastest-growing component, rising from 624.6 s (33.7%) at 3 nodes to 932.4 s (40.5%) at 9 nodes ($\beta = 50.5$ [42.1, 58.9]), accounting for 70% of the total growth and driven by its network configuration overhead scaling with cluster size. PDLC follows as the second largest, increasing from 429.0 s (23.1%) to 483.5 s (21.0%, $\beta = 8.6$ [3.9, 13.4]), while MDM grows moderately from 383.9 s (20.7%) to 427.3 s (18.5%, $\beta = 6.8$ [2.6, 11.0]). Preparation remains relatively stable, from 346.1 s (18.6%) to 389.2 s (16.9%, $\beta = 6.8$ [3.0, 10.6]), while SWM remains consistently minimal across all configurations, ranging from 72.3 s to 70.7 s (<4%, with no measurable growth $\beta = -0.3$ [−1.1, 0.4]). Figure 3c reports CODECO installation time overhead per component in a KinD environment (in INF1). For 3 nodes, the total CODECO installation requires 1380 s, increasing to 1554.0 s for 5 nodes (+12.6%), 2232.0 s for 10 nodes (+61.7%) and 3218.0 s for 20 nodes (+133.2% against 3 nodes, $\beta = 109.5$ s/node [86.6, 132.4]). In terms of components, NetMA is the dominant time-consuming factor with 44.5% at 3 nodes, reaching 63.6% at 20 ($\beta = 84.6$ [68.0, 101.3]), followed by MDM (24.3–23.2%, $\beta = 24.9$ [19.2, 30.5]) and PDLC (24.3–11.2%). Across Figure 3b,c, NetMA accounts for 70% of the per-node cost under CODEF and 77% under KinD, while SWM shows no measurable growth and PDLC grows by 8.6 and 2.0 s/node, respectively. Within the tested range, installation time growth stems from establishing secure, programmable pod-to-pod network paths rather than decision components, aligning with NetMA combining per-node switching and probing (Section 3). Slopes reflect installation time only, while KinD results include single-host container contention, not extending to runtime scalability performance.

5.3. Pod/Service Startup and Deletion Times

This section evaluates pod startup and deletion latency for three applications of increasing complexity (pause pods, dummy frontend-backend and Bookinfo), comparing CODECO against vanilla K8s across diverse infrastructure configurations and varying deployment scales. Absolute times differ across infrastructures due to hardware class, Kubernetes distribution and control-plane variations. Each comparison is therefore made

within an infrastructure, between the two configurations on the same hardware, so that the difference isolates the effect of CODECO, whereas the spread of the baseline values across infrastructures measures the effect of the infrastructure itself.

5.3.1. Application 1: Pod Readiness per Pod Batch

Figure 4 (top row) compares vanilla K8s against CODECO in terms of pod startup latency for batch deployments of *pause* pods (10–150). The added readiness latency grows with batch size and depends strongly on the infrastructure. Across the evaluated UCs, CODECO exhibits varying behaviour relative to vanilla K8s, with results depending on both batch size and underlying infrastructure. CODECO is slower on every infrastructure except INF4, and the gap grows with batch size, reaching 19.38 s (+51.4%) on INF2 at 150 pods. In INF2, mean readiness rises from 5.46 s (K8s: 3.98 s) at 10 pods to 57.11 s (K8s: 37.73 s) at 150 pods (+51.4%). In INF3, the overhead is 8.35 s (+15.4%) at 150 pods, while INF6 shows +5.71 s (+20.7%). On INF4, the two platforms are similar: at 150 pods CODECO reaches 46.96 s against 47.96 s for vanilla K8s ($\Delta = -1$ s, 95% CI $[-2.7, 0.7]$ s), within the run-to-run variability of both platforms. Finally, INF5 exhibits the highest readiness times, reflecting the constrained compute capacity of RPi workers. Vanilla K8s reaches readiness in 7.5 s (10 pods) and CODECO requires 10.5 s (+40%), while the difference reaches 12% (100 pods). The 150-pod batch was not deployed due to resource limitations.

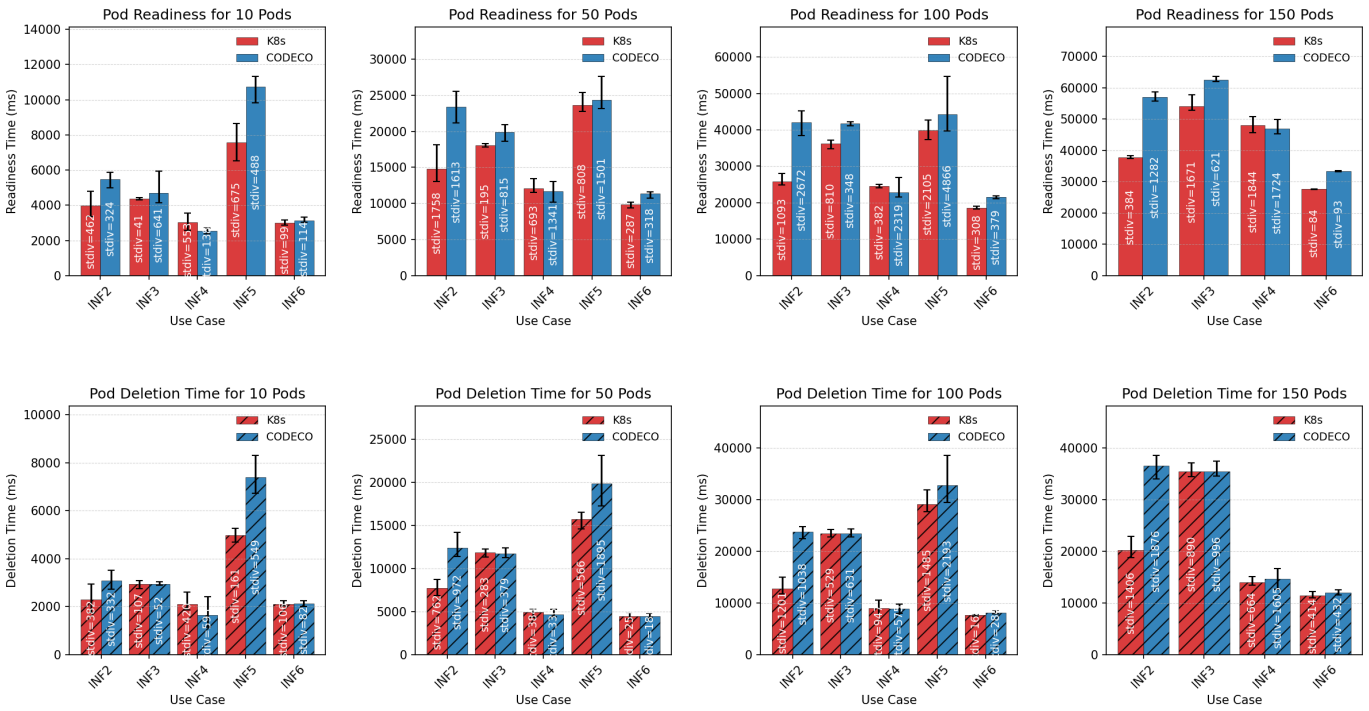


Figure 4. Mean pod readiness time (**top** row) and deletion time (**bottom** row) for a batch of pause pods (10–150). Error bars show the observed minimum to maximum. Standard deviation is reported in each bar.

Variability, as captured by standard deviation (*SD*), also differs across UCs and platforms. In INF2, vanilla K8s exhibits higher variability at 50 pods (1758 ms) compared to CODECO (1613 ms), whereas at 150 pods both platforms show reduced variance (K8s at 384 ms, while CODECO at 1282 ms). INF3 shows a wider spread for vanilla K8s (41–1671 ms across scales) than for CODECO (348–815 ms), while INF6 (AWS-based VMs) exhibits the most consistent behaviour, with *SD* below 400 ms. For INF5, the SD ranges between 488–2105 ms for CODECO, consistent with the constrained but stable nature of the RPi

hardware. The baseline values isolate the infrastructure effect as readiness times differ substantially across UCs due to infrastructure heterogeneity. At 50 pods under K8s, INF6 produces the fastest baseline (9.84 s at 50 pods), followed by INF4 (12.11 s) and INF2 (14.77 s), i.e., a factor of about two between the fastest x86 cloud VMs and the ARM-based RPi workers under K3s. Finally, INF5 exhibits the slowest baseline readiness (20 s at 50 pods for K8s), as a result of the ARM-based RPi workers operating under K3s.

In terms of pod deletion (Figure 4 (bottom row)), the overhead is small on most infrastructures but not uniform. At 150 pods, the mean difference is negligible on INF3 $[-0.9, 0.9]$ s, +0.7 s on INF4 $[-0.5, 1.9]$ s and +0.5 s on INF6 $[0.1, 0.9]$ s, making the two configurations' results almost identical. Conversely, on INF2 the difference reaches +16.2 s at 150 pods $[14.6, 17.8]$ s, and on INF5 +4.1 s at 50 pods $[2.8, 5.4]$ s and +3.6 s at 100 $[1.8, 5.4]$ s, the latter showing the highest infrastructure deletion times, consistent with the slower I/O and processing capabilities of RPi-based nodes. Therefore, deletion overhead is an order of magnitude smaller than the corresponding readiness on INF3 and INF6on INF3 and INF6, indicating that in this workload case CODECO primarily affects pod initialisation.

5.3.2. Application 2: Dummy Frontend-Backend

Figure 5 (top row) shows the readiness time of the dummy frontend-backend application while scaling the number of *frontend* replicas (1–50). Compared to Section 5.3.1, this captures service-level startup behaviour, i.e., the impact of deploying a structured application (multiple replicas and Kubernetes objects) under vanilla K8s against CODECO.

The CODECO interval here includes provisioning the declared service channel through NetMA, a step with no counterpart in the baseline, so part of the overhead reported below is the cost of that capability. The dummy application experiments also reveal substantial performance variation across infrastructures. In INF6, vanilla K8s scales from 4.22 s (1 replica) to 18.38 s (50 replicas) and CODECO from 5.67 s to 32.03 s, with absolute overheads ranging from 1.45 s to 13.65 s. In INF2, both configurations exhibit higher median times due to the remote, cloud-hosted control plane and the virtualisation overhead. K8s ranges from 5.44 s to 48.35 s, while CODECO fluctuates between 13.53–52.63 s. On INF2 the relative overhead narrows with replica count, from +150% (1 replica) to +8.7% (50 replicas), and on INF5 from +67% to +33%, although CODECO remains slower than the baseline on INF5 at every replica count. INF6, by contrast, moves in the opposite direction, from +34.4% (1 replica) to +144% (10 replicas), before settling at +75% and +74% at twenty-five and fifty. Whether the relative overhead shrinks with workload density is therefore infrastructure-dependent. INF6 exhibits low variability for vanilla K8s (223–338 ms) but higher variance for CODECO, particularly at 50 replicas (4217 ms). INF4 shows high variability (3.78–10.19 s, 1 replica), while CODECO maintains tighter distributions (15.64–15.74 s, 25 replicas).

In terms of deletion, Figure 5 (bottom row) reports the pod deletion times. On INF6, the mean values of vanilla K8s increase from 2.63 s (1 replica) to 18.42 s (50 replicas) and CODECO from 3.77 s to 18.92 s, with the largest gap at ten replicas (+6.27 s), narrowing to +0.5 s at fifty, where both configurations are comparable. INF2 and INF5 reverse at scale. On INF5, CODECO is 2 s slower at one and ten replicas, but faster by 1.3 s at twenty-five and 7 s at fifty. INF2 follows the same pattern, with CODECO faster by 5 s at twenty-five and 14.8 s at fifty. Therefore, unlike the isolated pods of Section 5.3.1, deletion overhead does not scale with replica count in the same manner as readiness, shifting from a performance penalty to an advantage at higher replica counts on the two infrastructures.

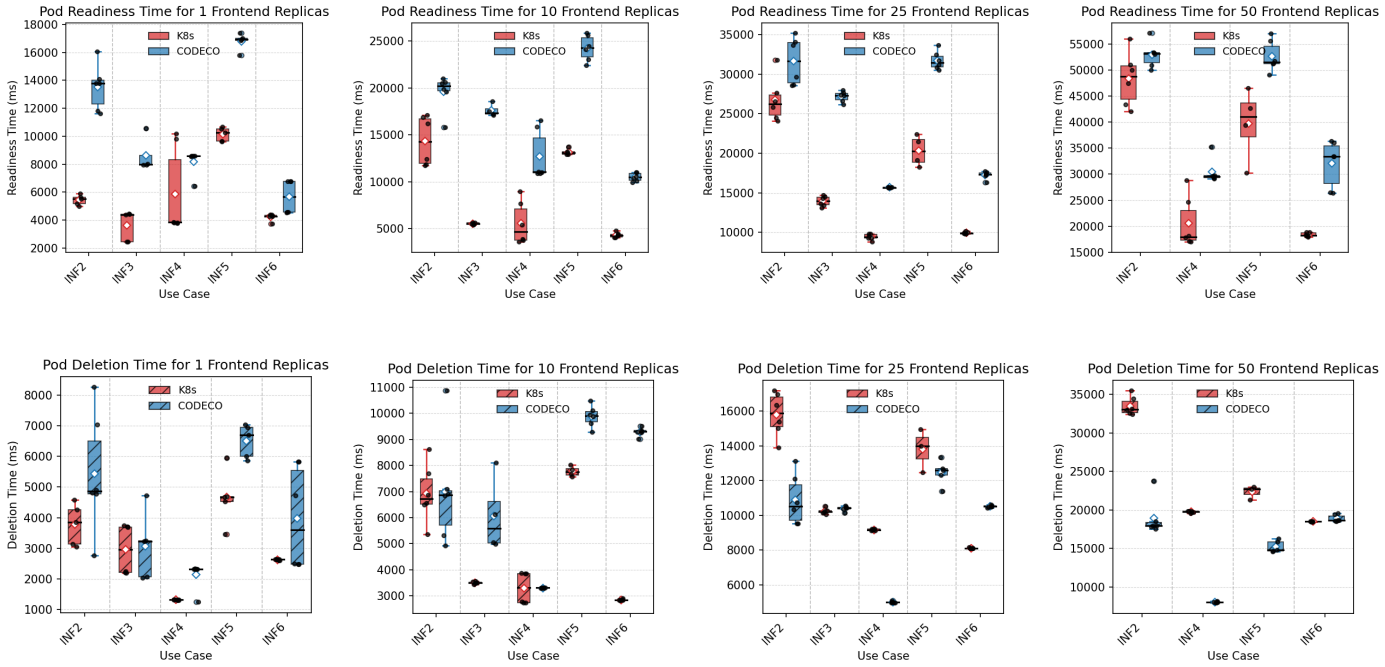


Figure 5. Boxplots of pod readiness time (**top** row) and deletion time (**bottom** row) for CODECO dummy (frontend-backend) app with varying frontend replicas (1–50). Boxplots show median (horizontal line), interquartile range (Q1–Q3), IQR (box), 1.5 × IQR whiskers and outliers.

5.3.3. Application 3: Bookinfo

Figure 6 shows the mean deployment (readiness) and deletion times for the Bookinfo application under vanilla K8s and CODECO. Across the evaluated UCs, CODECO introduces measurable startup overhead that varies by infrastructure. In INF6, K8s reaches readiness in 3.5 s on average, while CODECO requires 11 s. INF3 exhibits similar behaviour with vanilla K8s completed in 3 s against CODECO at 13 s, while INF2 shows K8s at 4.5 s and CODECO at 14 s. INF4 presents the smallest gap, with K8s at 6.5 s and CODECO at 9 s. Finally, INF5 exhibits the largest readiness times among all UCs, with K8s at approximately 20 s and CODECO at 30 s, reflecting the resource constraints of RPi-based edge nodes.

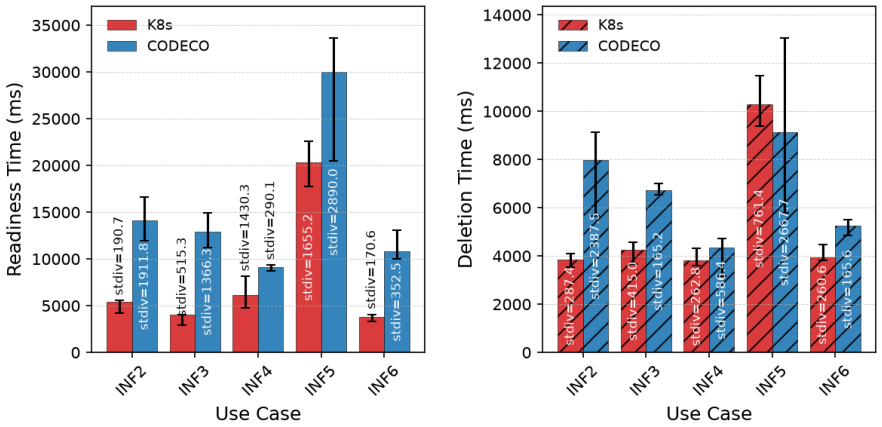


Figure 6. Mean deployment (**left**) and deletion times (**right**) for the Bookinfo app. Error bars show the observed minimum to maximum. Standard deviation is reported in each bar.

For deletion, CODECO's overhead ranges from +0.60 s in INF4 (+16.2%) to +4.3 s in INF2 (+116.2%). Excluding INF5, vanilla K8s completes teardown in 3.7–4.2 s, while CODECO requires 4.3–8 s. Regarding variability, vanilla K8s maintains stable performance

(261–415 ms), while CODECO ranges from 165 ms for INF3 and INF6, to higher variance in INF2 (2388 ms). Similarly to readiness results, INF5 exhibits the highest baseline deletion time (10.5 s vs. 9.5 s). Despite the overhead compared to previous experiments and applications, the absolute times remain below 15 s for startup and below 10 s for deletion on all infrastructures except INF5, where readiness reaches 30 s and baseline deletion 10.5 s.

5.4. Resource Footprint Metrics

Figure 7 and Table 7 report the resource footprint (CPU, memory and power) for master and worker nodes. The clusters are deployed with CODEF, comparing vanilla K8s against CODECO. Once the cluster reached a stable state, metrics were gathered over a 30-min period, with the final values representing the mean performance across this interval (cf. Table 7). Under idle conditions, CPU utilisation increases on every node following CODECO deployment, due to its orchestration components. The master node reports 4.66% (+2.9 pp), while workers reach 3.81% and 2.3% (+2.71 pp and +1.61 pp). The sum of per-node utilisation percentages rises from 3.55 to 10.77, corresponding to an increase of 7.22 percentage points. CODECO reports higher variability, reaching 15% on workers, compared to K8s's stable range of 0.58–2.26%.

Table 7. Average node-level resource footprint and estimated power consumption for vanilla K8s and CODECO deployments in INF6 (3-node cluster in idle state).

Node	CPU (%) †		Mem (GB)		Power (W) *	
	K8s	CODECO	K8s	CODECO	K8s	CODECO
Master	1.76	4.66	1.15	3.14	36.3	38.3
Worker 1	1.10	3.81	0.99	2.92	35.8	37.8
Worker 2	0.69	2.30	0.57	1.22	35.5	36.6
Sum	3.55	10.77	2.71	7.28	107.6	112.7
Overhead	+7.22 pp		+4.57		+5.1 (+4.7%)	

* Power estimation parameters (Intel 8259CL): $P_{idle} = 35$ W, $P_{max} = 95$ W per node. † CPU values are per-node (in %).

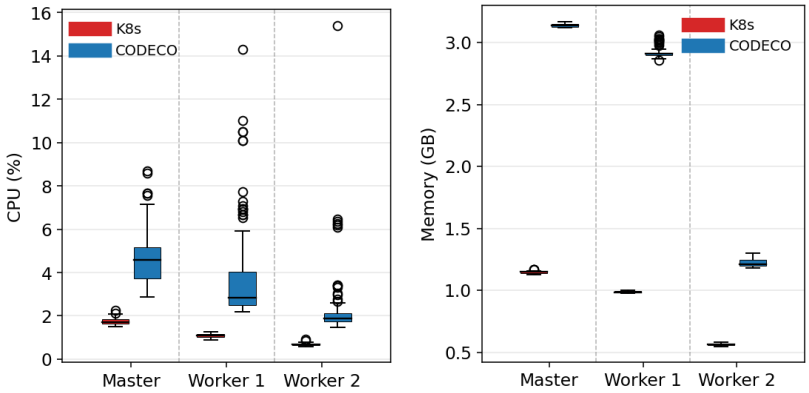


Figure 7. (i) CPU utilisation (%) and (ii) memory usage (GB) of master and worker nodes for vanilla K8s and CODECO in idle state with CODEF for INF6. Boxplots show median (horizontal line), interquartile range (Q1–Q3), IQR (box), 1.5 × IQR whiskers and outliers.

The memory overhead is more notable as CODECO significantly expands the cluster's resource requirements. While vanilla K8s deployment maintains a footprint for the master at 1.15 GB and workers at 0.99 GB and 0.57 GB, CODECO increases memory consumption. The master node rises to 3.14 GB (+1.99 GB), while the workers reach 2.92 GB and 1.22 GB,

respectively. These per-node increases are of the same order as the entire idle memory of a lightweight Kubernetes control plane for K3s, k0s and MicroK8s [6].

In terms of power, we estimate the draw of each node with the established linear power model (Table 7). Under idle conditions, the estimated total rises from 107.6 W (vanilla K8s) to 112.7 W with CODECO, i.e., $\Delta\hat{P} \approx 5$ W (3.8–7.6 W under the sensitivity ranges of Section 4), a 4.7% increase of the estimated total that depends on the assumed idle power. Similarly, Figure 8 illustrates the system-level resource footprint (CPU, memory) for KinD deployments, comparing KinD against KinD + CODECO as the cluster scales from 3 to 20 nodes. Unlike CODEF, where each node runs on a dedicated VM, KinD deploys all nodes as containers on a single host, resulting in aggregated resource consumption. In idle state, KinD presents low CPU utilisation, ranging from 0.6% (3 nodes) to 3.1% (20 nodes). CODECO increases this consumption to 2.3–7.5%. Memory footprint follows a similar trend with KinD consuming 4.91–6.28 GB, while CODECO requires 12.18–17.70 GB, representing an overhead of +7.27 GB (3 nodes) to +11.42 GB (20 nodes). In terms of power (cf. Table 8), KinD consumes 64.5 W (20 nodes), while CODECO 71.4 W, a 10.7% overhead.

Table 8. Average system-level resource footprint and estimated power consumption for KinD and KinD + CODECO deployments in INF1 (scaling nodes, idle state).

Nodes	CPU (%)		Mem (GB)		Power (W) *	
	KinD	CODECO	KinD	CODECO	KinD	CODECO
3	0.6	2.3	4.91	12.18	61.6	64.9
5	1.8	2.9	4.98	12.92	62.9	65.6
10	2.4	4.2	5.42	14.88	63.6	67.4
20	3.1	7.5	6.28	17.70	64.5	71.4
Max Overhead	+4.4 pp		+11.42		+6.9 (+10.7%)	

* Power estimation parameters (i9-10900K): $P_{idle} = 60$ W, $P_{max} = 165$ W.

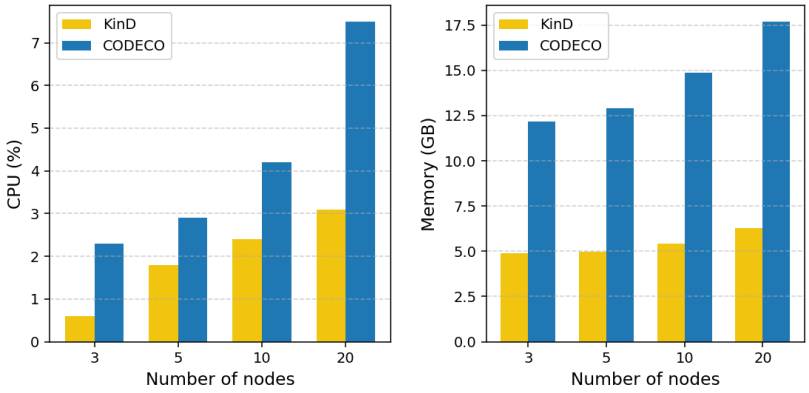


Figure 8. Average system (i) CPU utilisation (%) and (ii) memory usage (GB) for KinD and KinD + CODECO deployments in idle state for INF1.

Following the results of Figure 3, Table 9 decomposes idle-state CPU consumption across the CODECO components and extends the comparison to cluster-level CPU and memory under sustained Bookinfo load. The idle-state mirrors the installation-time results: NetMA accounts for 80.2% of the framework's CPU footprint at rest, while MDM follows at 18.4%, reflecting its co-installed tools (Kafka, Zookeeper, Neo4j). PDLC, SWM and ACM each remain below 1%. Under sustained load, CPU overhead falls as request rate increases compared to vanilla K8s, from +68.3% at 25 RPS to +54.7% and +31.5% at 50 and 100 RPS, respectively, since the CODECO components' consumption stays close to its idle value while application's CPU demand grows around it. Memory overhead shows the opposite pattern,

remaining at +165%, confirming it does not scale with request volume. Decomposing the idle footprint by component shows that CPU and memory are dominated by different roles. NetMA accounts for 80.2% of the layer's CPU but only 22.2% of its memory, while MDM accounts for 18.4% of CPU and 64.3% of memory. The latter is concentrated in MDM's backing services: Neo4j (1.05 GB), Kafka (0.965 GB) and Zookeeper (0.179 GB) accounting for 2.193 GB, i.e., 53.6% of the layer's memory against 10.6% of its CPU. Since the heaps and caches of these services are sized by configuration, this explains why the memory overhead remains near +165% across all three request rates while the CPU overhead falls. NetMA's own memory is concentrated in the L2S-M controller (0.667 GB). These shares are computed over container working-set memory in the CODECO namespaces, a total of 4.09 GB, while the 4.57 GB of Table 7 is the node-level increase. The 0.48 GB difference is the residual kubelet and container-runtime bookkeeping for the additional pods and kernel state for the overlay network.

Table 9. Platform CPU and memory of Bookinfo load and per-component idle footprint for INF6.

	25 RPS	**50 RPS**	**100 RPS**		
CPU (cores)					
K8s platform	0.377	0.436	0.550		
CODECO platform	0.634	0.674	0.723		
Overhead	+68.3%	+54.7%	+31.5%		
Memory (GB)					
K8s platform	2.72	2.71	2.75		
CODECO platform	7.25	7.23	7.26		
Overhead	+166.5%	+166.8%	+164.0%		
	NetMA	**MDM**	**PDLC**	**SWM**	**ACM**
Idle CPU share †	80.2%	18.4%	0.55%	0.45%	0.4%
Idle memory share †	22.2%	64.3%	12.9%	0.22%	0.28%

† CPU and memory shares are computed over the pods in the CODECO namespaces, not over the node-level increase of Table 7. Neo4j 1.050 GB, Kafka 0.965 GB and Zookeeper 0.179 GB together account for 2.193 GB, i.e., 53.6% of the layer's memory but only 10.6% of its CPU. NetMA's memory is concentrated in the L2S-M controller, at 0.667 GB.

6. Discussion, Limitations and Future Work

Network Enforcement is the main contributor to installation time and idle CPU consumption. The component breakdown overhead (Table 9 and Figure 3) show that the network enforcement component (NetMA) is the dominant and scale-sensitive contributor to installation time, accounting for 34–64% depending on cluster size and environment, while SWM shows no measurable growth and PDLC grows slowly in installation time. The same holds at idle, where NetMA accounts for 80.2% of the framework's CPU, against 18.4% for data observability (MDM) and below 1% each for SWM, PDLC and ACM (Table 9). However, memory is dominated by MDM which accounts for 64.3% of the layer's memory (more than half of which resides in its sized backing services) compared to 22.2% for NetMA, showing that the two resources have distinct deployment boundaries. Under load, memory overhead remains near +165% while CPU falls from +68.3% (25 RPS) to +31.5% (100 RPS), indicating a fixed cost, which is significant for IoT deployments. We argue that enforcing application-intent connectivity, as declared in the CAM service channels (Table 3), must establish and maintain programmable per-node network paths, an overhead attributed to the nature of the capability itself. Consequently, within the tested range, network configuration drives installation cost more than the cognitive layer, scaling with the underlying infrastructure. This extends beyond CODECO as application-level connectivity requires

per-node state, e.g., in eBPF CNIs (per-node agents), service meshes (per-pod proxies) and telemetry stacks (per-node exporters) [35–37]. Conversely, decision logic can be centralised and distributed as configuration, explaining the minimal installation-time growth observed (SWM, PDLC).

CEI deployment and trade-offs. Intelligent decision-making and automation come with costs, in our case, memory consumption. The sum of per-node CPU utilisation rises from 3.55 to 10.77 (7.23 percentage points), and the estimated power overhead of 4.7% is comparable to figures reported for the observability and adaptive networking layers that CEI deployments already provision [37]. The resource limits exposed by our experiments are the conditions cognitive scheduling is designed to manage. Several high-density configurations reached the resource limits of their host clusters, such as the 150-pod deployment on the constrained INF5 cluster. In an unmanaged deployment, such limits surface as scheduling failures or degraded service, while a cognitive orchestrator accommodates them through placement decisions that account for per-node capacity. Baseline readiness time varies by a factor of two across infrastructures, while cross-configuration ratios span 0.98 to 1.51 at identical scale. Hardware class and orchestration layer thus contribute effects of comparable magnitude.

Implications for IoT deployments. The cost profile observed here is particularly relevant to IoT pipelines, which typically run continuously and whose memory demand for stream buffers, inference models and local time series often determines what an edge node can host. If the per-worker memory increase measured on INF6 (0.65 to 1.93 GB) carried over to a 4 GB Raspberry Pi 5 node (INF3), the layer would occupy 16 to 48% of node memory prior to service deployment, though this projection requires physical verification. This spread reflects component decomposition as cluster-wide data-observability backends reside on worker 1 (1.93 GB), making the 0.65 GB footprint of worker 2 representative of the true recurring per-node overhead. Second, the observed installation-time and idle CPU contribution of NetMA suggests that selective network enforcement could be investigated as a means of reducing resource overhead. Finally, the relative overhead falls with workload density on INF2 and INF5 but does not fall monotonically on others (INF6).

Generalisability. The results here are specific to CODECO, but the models of Section 4 apply to any layer whose components can be listed as workloads. A component deployed once per node adds a cost that grows linearly with cluster size, in contrast to a per-cluster component, whether or not it implements decision logic. Prior work is consistent with this, since a service mesh charges its enforcement cost to each pod [36] and frameworks that run a decision agent or ML model on every node, such as decentralised multi-agent schedulers [16,17], also make their decision layer grow with the node count. Stateful backends such as message brokers and graph databases use memory set mostly by configuration and not by load, so similar backends should require a similar memory cost.

Limitations and future work. The evaluation operates within specific methodological boundaries, as current resource attribution is performed at the namespace level, separating CODECO's components from the application. Subcomponent analysis under load, or measurements regarding rescheduling and workload migration fall outside our evaluation scope. Additionally, the reported reduction in manual interventions is derived from standard documented procedures and our defined action-count metric, therefore, the observed automation gains may vary depending on action granularity and the existing target-environment workflows. To complement the current controlled request–response benchmark (Bookinfo), future work will incorporate variable arrival rates and publish–subscribe telemetry ingestion typical of sensor fleets, reporting loaded footprint which reflects the layer's overhead under query traffic against monitoring services. In addition, power metrics are modelled, as most infrastructures lack hardware power instrumentation, an infrastructure limitation highlighted in Section 4.2.3. Future work will evaluate the

runtime overhead of SWM and PDLC under dynamic deployments beyond installation time and single-host KinD limits, while jointly assessing cognitive decision-making benefits in terms of QoS and fault recovery performance. Benchmarking against alternative cognitive orchestration frameworks remains a separate work, given that per-role resource attribution depends on modularly deployable components (Table 2). Finally, we plan to quantify the net cost-benefit balance of rescheduling and migration, integrate MQTT/time-series IoT ingestion workloads and extend the evaluation to federated multi-cluster environments.

7. Conclusions

This paper proposed a component-level cost-attribution methodology for cognitive orchestration layers and applied it to the open-source CODECO framework across six heterogeneous infrastructures, including operational IoT deployments and three K8s distributions, measuring deployment and lifecycle latency, manual intervention, and resource and estimated power footprint against vanilla K8s. The study characterises the cost of the layer against the automation it provides, with the principal findings as follows:

- **Automation gains are substantial and originate mainly in declarative provisioning.** CODEF and CODECO reduce manual intervention by 84.2% for cluster deployment, 90.5% for Kubernetes installation and 81.8% for service deployment, an aggregate reduction of 87.5%. The majority arises in the provisioning and installation phases automated by declarative tooling.
- **Network enforcement accounts for the largest share of measured installation time and idle CPU usage.** Network management accounts for 34–64% of installation time and 80.2% of the layer's idle CPU and drives its scaling behaviour, while the scheduling and learning components grow little in installation time.
- **Memory is the binding constraint at the edge.** At idle, the layer adds 7.22 percentage points of summed CPU utilisation, 4.57 GB of memory and an estimated +5 W (4.7% of the estimated total) on a three-node cluster, and 4.4 pp, 11.42 GB and 10.7% on a twenty-node KinD cluster. Unlike CPU, which is dominated by network enforcement, 64.3% of the layer's memory is attributable to data observability and over half of it to statically sized backing services. Per-node memory increases are of the same order as the entire idle footprint of a lightweight K8s distribution [6].
- **Relative overhead is infrastructure-dependent.** For the two-service application on INF2, the relative latency overhead falls from +150% at a single replica to +8.7% at fifty, and INF5 narrows similarly, from +67% to +33%. On INF6, however, the overhead does not fall with replica count, so the relationship between workload density and relative overhead depends on the infrastructure.
- **Cost is strongly infrastructure-dependent.** Pod readiness for 150-pod batches ranged from 33.3 s on INF6 to 62.6 s on the Raspberry Pi-based INF3, while additional latency ranged from statistically indistinguishable (INF4) to 19.4 s (INF2). High-replica deployments exposed resource limits on certain clusters, underlining the value of the resource-aware scheduling CODECO natively supports.

Author Contributions: Conceptualization, G.K., V.T. and R.C.S.; methodology, G.K., I.D. and R.C.S.; software, G.K., I.D., J.L., F.W., D.U., G.S., M.A.P.V., J.S., P.K. and R.C.S.; validation, G.K., I.D., J.L., F.W., D.U., G.S., M.A.P.V., J.S., P.K. and R.C.S.; formal analysis, G.K., I.D. and V.T.; investigation, G.K., J.L., D.U., M.A.P.V., P.K. and R.C.S.; resources, G.K., J.L., F.W., G.S., J.S., P.K. and R.C.S.; data curation, G.K. and R.C.S.; writing—original draft preparation, G.K., V.T. and R.C.S.; writing—review and editing, G.K., I.D., V.T. and R.C.S.; visualization, G.K. and I.D.; supervision, V.T. and R.C.S.; project administration, V.T. and R.C.S.; funding acquisition, V.T. and R.C.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been funded by the European Commission in the context of the Horizon Europe CODECO project, under grant number 101092696, and by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 23.00028.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All experimental artefacts supporting the reported results, including deployment scripts, configuration files, experiment specifications and raw measurement data, are openly available at https://github.com/gkoukis/CODECO-paper-kp1.1-2 (accessed on 28 October 2026). The CODECO source code is available at https://gitlab.eclipse.org/eclipse-research-labs/codeco-project (accessed on 28 October 2026), including the installation script (https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/acm/-/blob/main/scripts/post_deploy.sh (accessed on 28 October 2026)), the application-model definition (https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/acm/-/blob/main/config/crd/bases/codeco.he-codeco.eu_codecoapps.yaml, (accessed on 28 October 2026)) and the sample two-service application (https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/acm/-/blob/main/config/samples/codeco_v1alpha1_codecoapp_ver3.yaml, (accessed on 28 October 2026)).

Conflicts of Interest: Author Rute C. Sofia was employed by the company Fortiss GmbH. The remaining author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

ACM	Automated Configuration Management
BMWG	Benchmarking Methodology Working Group
CAM	CODECO Application Model
CEI	Cloud–Edge–IoT
CI	Confidence Interval
CNI	Container Network Interface
CODEF	CODECO Experimentation Framework
CRD	Custom Resource Definition
GNN	Graph Neural Network
IETF	Internet Engineering Task Force
K8s	Kubernetes
KPI	Key Performance Indicator
MDM	Metadata Manager
MIP	Manual Intervention Percentage
NetMA	Network Management and Adaptation
PDLC	Privacy-preserving Decentralised Learning and Context-awareness
QoS	Quality of Service
SDN	Software-Defined Networking
SWM	Scheduling and Workload Migration
UC	Use Case

References

1. Witczak, D.; Szymoniak, S. Review of Monitoring and Control Systems Based on Internet of Things. *Appl. Sci.* **2024**, *14*, 8943. [CrossRef]
2. Khan, L.U.; Yaqoob, I.; Tran, N.H.; Kazmi, S.M.A.; Dang, T.N.; Hong, C.S. Edge-Computing-Enabled Smart Cities: A Comprehensive Survey. *IEEE Internet Things J.* **2020**, *7*, 10200–10232. [CrossRef]
3. Arthurs, P.; Gillam, L.; Krause, P.; Wang, N.; Halder, K.; Mouzakitis, A. A Taxonomy and Survey of Edge Cloud Computing for Intelligent Transportation Systems and Connected Vehicles. *IEEE Trans. Intell. Transp. Syst.* **2022**, *23*, 6206–6221. [CrossRef]
4. Li, J.; Gu, C.; Xiang, Y.; Li, F. Edge-cloud Computing Systems for Smart Grid: State-of-the-art, Architecture, and Applications. *J. Mod. Power Syst. Clean Energy* **2022**, *10*, 805–817. [CrossRef]
5. Qiu, T.; Chi, J.; Zhou, X.; Ning, Z.; Atiquzzaman, M.; Wu, D.O. Edge Computing in Industrial Internet of Things: Architecture, Advances and Challenges. *IEEE Commun. Surv. Tutor.* **2020**, *22*, 2462–2488. [CrossRef]

6. Koziolek, H.; Eskandani, N. Lightweight Kubernetes Distributions: A Performance Comparison of MicroK8s, k3s, k0s, and Microshift. In Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering, Coimbra, Portugal, 15–19 April 2023; pp. 17–29. [CrossRef]
7. Čilić, I.; Krivić, P.; Podnar Žarko, I.; Kušek, M. Performance Evaluation of Container Orchestration Tools in Edge Computing Environments. *Sensors* **2023**, *23*, 4008. [CrossRef]
8. Rejiba, Z.; Chamanara, J. Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches. *ACM Comput. Surv.* **2022**, *55*, 1–37. [CrossRef]
9. Carrión, C. Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges. *ACM Comput. Surv.* **2022**, *55*, 1–37. [CrossRef]
10. Volcano: Cloud Native Batch Scheduling System for Compute-Intensive Workloads. CNCF Project. Available online: https://volcano.sh (accessed on 28 August 2026).
11. Koordinator: QoS-Based Scheduling for Colocating Workloads on Kubernetes. CNCF Project. Available online: https://koordinator.sh (accessed on 28 August 2026).
12. Xiong, Y.; Sun, Y.; Xing, L.; Huang, Y. Extend Cloud to Edge with KubeEdge. In Proceedings of the 2018 IEEE/ACM Symposium on Edge Computing (SEC), Seattle, WA, USA, 25–27 October 2018; pp. 373–377. [CrossRef]
13. OpenYurt: Extending Your Native Kubernetes to Edge. Available online: https://openyurt.io (accessed on 29 September 2026).
14. Iorio, M.; Risso, F.; Palesandro, A.; Camiciotti, L.; Manzalini, A. Computing Without Borders: The Way Towards Liquid Computing. *IEEE Trans. Cloud Comput.* **2023**, *11*, 2820–2838. [CrossRef]
15. Toka, L.; Dobreff, G.; Fodor, B.; Sonkoly, B. Machine Learning-Based Scaling Management for Kubernetes Edge Clusters. *IEEE Trans. Netw. Serv. Manag.* **2021**, *18*, 958–972. [CrossRef]
16. Ma, Y.; Zhao, Y.; Hu, Y.; He, X.; Feng, S. Multi-Agent Deep Reinforcement Learning for Joint Task Offloading and Resource Allocation in IIoT with Dynamic Priorities. *Sensors* **2025**, *25*, 6160. [CrossRef]
17. Zhu, J.; Chen, H.; Zhang, J.; Guo, F.; Zomaya, A.Y.; Yang, R. HiRL: Hierarchical Reinforcement Learning for Coordinated Resource Management in Heterogeneous Edge Computing. *arXiv* **2026**, arXiv:2605.10443.
18. Böhm, S.; Wirtz, G. Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes. In Proceedings of the 13th Central European Workshop on Services and Their Composition (ZEUS 2021), Bamberg, Germany, 25–26 February 2021; pp. 65–73.
19. Sofia, R.C.; Salomon, J.; Ferlin-Reiter, S.; Garces-Erice, L.; Urbanetz, P.; Mueller, H.; Touma, R.; Espinosa, A.; Contreras, L.M.; Theodorou, V.; et al. A Framework for Cognitive, Decentralized Container Orchestration. *IEEE Access* **2024**, *12*, 79978–80008. [CrossRef]
20. Koukis, G.; Skaperas, S.; Kapetanidou, A.; Mamatas, L.; Tsaoussidis, V. An Open-Source Experimentation Framework for the Edge Cloud Continuum. In Proceedings of the IEEE INFOCOM 2024—IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Vancouver, BC, Canada, 20 May 2024. [CrossRef]
21. Santos, J.; Wauters, T.; Volckaert, B.; De Turck, F. Resource Provisioning in Fog Computing: From Theory to Practice. *Sensors* **2019**, *19*, 2238. [CrossRef]
22. Santos, J.; Tantawi, A.; Maniotis, P.; Wang, C.; Tardieu, O.; Wauters, T.; De Turck, F. Sakkara: Intelligent Topology-Aware Scheduling for Kubernetes in the Age of AI. *IEEE Trans. Netw. Serv. Manag.* **2026**, *23*, 5437–5451. [CrossRef]
23. C. Sofia, R.; Dykeman, D.; Urbanetz, P.; Galal, A.; Dave, D.A. Dynamic, Context-Aware Cross-Layer Orchestration of Containerized Applications. *IEEE Access* **2023**, *11*, 93129–93150. [CrossRef]
24. Mohajer, A.; Hajipour, J.; Leung, V.C.M. Dynamic Offloading in Mobile Edge Computing With Traffic-Aware Network Slicing and Adaptive TD3 Strategy. *IEEE Commun. Lett.* **2025**, *29*, 95–99. [CrossRef]
25. Mohajer, A.; Mirzaei, A.; Darabi, M.; Fernando, X. Joint SLA-Aware Task Offloading and Adaptive Service Orchestration With Graph-Attentive Multi-Agent Reinforcement Learning. *IEEE Trans. Netw. Serv. Manag.* **2026**, *23*, 3737–3754. [CrossRef]
26. Wang, L.; Lu, K.; Zhang, N.; Qu, X.; Wang, J.; Wan, J.; Li, G.; Xiao, J. Shoggoth: Towards Efficient Edge-Cloud Collaborative Real-Time Video Inference via Adaptive Online Learning. In Proceedings of the 2023 60th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 9–13 July 2023; pp. 1–6. [CrossRef]
27. Zhang, Z.; Zhao, Y.; Li, H.; Lin, C.; Liu, J. DVFO: Learning-Based DVFS for Energy-Efficient Edge-Cloud Collaborative Inference. *IEEE Trans. Mob. Comput.* **2024**, *23*, 9042–9059. [CrossRef]
28. Nizamis, A.; Neises, J.; Ospina, D.; Wajid, U.; Valero López, C.I.; Trakadas, P.; Votis, K.; Lazaro, O.; Nechifor, S.; Palau, C. ENACT: Adaptive Scheduling and Deployment of Data-Intensive Workloads on Energy-Efficient Edge-to-Cloud Continuum. In Proceedings of the 2024 IEEE International Conference on Engineering, Technology, and Innovation (ICE/ITMC), Funchal, Portugal, 24–28 June 2024; pp. 1–9. [CrossRef]
29. Townend, P.; Martí, A.P.; de la Iglesia, I.; Matskanis, N.; Ohlson Timoudas, T.; Hallmann, T.; Lalaguna, A.; Swat, K.; Renzi, F.; Bochoński, D.; et al. COGNIT: Challenges and Vision for a Serverless and Multi-Provider Cognitive Cloud-Edge Continuum. In Proceedings of the 2023 IEEE International Conference on Edge Computing and Communications (EDGE), Chicago, IL, USA, 2–8 July 2023; pp. 12–22. [CrossRef]

30. Cicconetti, C.; Carlini, E.; Paradell, A. EDGELESS Project: On the Road to Serverless Edge AI. In Proceedings of the 3rd Workshop on Flexible Resource and Application Management on the Edge, Orlando, FL, USA, 20 June 2023; FRAME '23; pp. 41–43. [CrossRef]
31. Katsaragakis, M.; Filippopoulos, O.; Sad, C.; Masouros, D.; Spatharakis, D.; Dimolitsas, I.; Filinis, N.; Zafeiropoulos, A.; Siozios, K.; Soudris, D.; et al. Multi-Partner Project: Orchestrating Deployment and Real-Time Monitoring—NEPHELE Multi-Cloud Ecosystem. In Proceedings of the 2025 Design, Automation & Test in Europe Conference (DATE), Lyon, France, 31 March–2 April 2025; pp. 1–6. [CrossRef]
32. Loaiza, M.; Savaglio, C.; Gravina, R.; Chatzopoulos, D.; Lalis, S. Agents in the Computing Continuum: The MLSysOps Perspective. In Proceedings of the 2023 International Conference on Embedded Wireless Systems and Networks, Rende, Italy, 25–27 September 2023; pp. 321–326.
33. Zhu, H.; Samizadeh, T.; Sofia, R.C. A CODECO Case Study and Initial Validation for Edge Orchestration of Autonomous Mobile Robots. *Computer* **2026**, *59*, 33–44. [CrossRef]
34. Ali, D.; Sofia, R.C. Experimenting with Energy-Awareness in Edge-Cloud Containerized Application Orchestration. *arXiv* **2025**, arXiv:2511.09116.
35. Qi, S.; Kulkarni, S.G.; Ramakrishnan, K.K. Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability. *IEEE Trans. Netw. Serv. Manag.* **2021**, *18*, 656–671. [CrossRef]
36. Zhu, X.; She, G.; Xue, B.; Zhang, Y.; Zhang, Y.; Zou, X.K.; Duan, X.; He, P.; Krishnamurthy, A.; Lentz, M.; et al. Dissecting Overheads of Service Mesh Sidecars. In Proceedings of the 2023 ACM Symposium on Cloud Computing, Santa Cruz, CA, USA, 30 October–1 November 2023; pp. 142–157. [CrossRef]
37. Dinga, M.; Malavolta, I.; Giamattei, L.; Guerriero, A.; Pietrantuono, R. An Empirical Evaluation of the Energy and Performance Overhead of Monitoring Tools on Docker-Based Systems. In *Proceedings of the Service-Oriented Computing (ICSOC 2023)*; Springer: Berlin/Heidelberg, Germany, 2023; pp. 181–196. [CrossRef]
38. Dermentzis, I.; Koukis, G.; Tsaoussidis, V. Trade-Offs in Kubernetes Security and Energy Consumption. *Urban Sci.* **2026**, *10*, 81. [CrossRef]
39. Amaral, M.; Chen, H.; Chiba, T.; Nakazawa, R.; Choochotkaew, S.; Lee, E.K.; Eilam, T. Kepler: A Framework to Calculate the Energy Consumption of Containerized Applications. In Proceedings of the 2023 IEEE 16th International Conference on Cloud Computing (CLOUD), Chicago, IL, USA, 2–8 July 2023; pp. 69–71. [CrossRef]
40. Pijnacker, B.; Setz, B.; Andrikopoulos, V. Container-level Energy Observability in Kubernetes Clusters. In Proceedings of the 2025 11th International Conference on ICT for Sustainability (ICT4S), Dublin, Ireland, 9–13 June 2025; pp. 69–79. [CrossRef]
41. Ruiu, P.; Lagorio, A.; Rubattu, C.; Anedda, M.; Sanna, M.; Fadda, M. Edge-to-Cloud Continuum Orchestrator Based on Heterogeneous Nodes for Urban Traffic Monitoring. *Future Internet* **2025**, *17*, 574. [CrossRef]
42. Jayachandra, A.B.; Suhas, S.; Alkhayyat, A.; Shaik, A.L.H.P.; Paramesh, S. Energy efficient mobile edge computing framework based on secure host level management for resource allocation. *Int. J. Cogn. Comput. Eng.* **2026**, *7*, 300–309. [CrossRef]
43. Tian, K. An Autonomous Energy-Aware Resource Scheduling Framework for Edge Computing Environments. In *Artificial Intelligence Review*; Springer: Berlin/Heidelberg, Germany, 2026. [CrossRef]
44. Kjorveziroski, V.; Filiposka, S. Kubernetes Distributions for the Edge: Serverless Performance Evaluation. *J. Supercomput.* **2022**, *78*, 13728–13755. [CrossRef]
45. Yang, J.; Saad, Z.; Wu, J.; Niu, X.; Leung, H.; Drew, S. A Survey on Task Scheduling in Carbon-Aware Container Orchestration, 2025. *arXiv* **2025**, arXiv:2508.05949.
46. Centofanti, C.; Santos, J.; Gudepu, V.; Kondepu, K. Impact of Power Consumption in Containerized Clouds: A Comprehensive Analysis of Open-Source Power Measurement Tools. *Comput. Netw.* **2024**, *245*, 110371. [CrossRef]
47. Open Cluster Management. 2024. Available online: https://open-cluster-management.io/ (accessed on 23 October 2026).
48. IETF. Application-Layer Traffic Optimization (ALTO) Working Group. Available online: https://datatracker.ietf.org/wg/alto/ (accessed on 25 August 2026).
49. Morton, A. Considerations for Benchmarking Virtual Network Functions and Their Infrastructure. RFC 8172, 2017. Available online: https://www.rfc-editor.org/info/rfc8172/ (accessed on 28 October 2026).
50. Tahhan, M.; O'Mahony, B.; Morton, A. Benchmarking Virtual Switches in the Open Platform for NFV (OPNFV). RFC 8204, 2017. Available online: https://www.rfc-editor.org/info/rfc8204/ (accessed on 28 October 2026).
51. Papadopoulos, A.V.; Versluis, L.; Bauer, A.; Herbst, N.; Kistowski, J.V.; Ali-Eldin, A.; Abad, C.L.; Amaral, J.N.; Tůma, P.; Iosup, A. Methodological Principles for Reproducible Performance Evaluation in Cloud Computing. *IEEE Trans. Softw. Eng.* **2021**, *47*, 1528–1543. [CrossRef]
52. Kubernetes SIGs. KinD: Kubernetes in Docker. Available online: https://kind.sigs.k8s.io (accessed on 25 August 2026).
53. Khan, K.N.; Hirki, M.; Niemi, T.; Nurminen, J.K.; Ou, Z. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* **2018**, *3*, 1–26. [CrossRef]

54. Fan, X.; Weber, W.D.; Barroso, L.A. Power Provisioning for a Warehouse-sized Computer. In Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07), San Diego, CA, USA, 9–13 June 2007; pp. 13–23. [CrossRef]
55. Cloud Carbon Footprint: Methodology. Available online: https://www.cloudcarbonfootprint.org/docs/methodology/ (accessed on 22 October 2026).
56. Istio. Bookinfo Application. Available online: https://istio.io/latest/docs/examples/bookinfo/ (accessed on 25 August 2026).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.