Future Generation Computer Systems 166 (2025) 107680

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

HephaestusForge: Optimal microservice deployment across the Compute Continuum via Reinforcement Learning

José Santos [a,*], Mattia Zaccarini [b], Filippo Poltronieri [b], Mauro Tortonesi [b], Cesare Stefanelli [b], Nicola Di Cicco [c], Filip De Turck [a]

[a] Ghent University - imec, IDLab, Department of Information Technology, Technologiepark - Zwijnaarde 126, Gent, 9052, Belgium
[b] Distributed Systems Research Group, University of Ferrara, Ferrara, Italy
[c] Department of Electronics, Information, and Bioengineering (DEIB), Politecnico di Milano, Milan, Italy

ARTICLE INFO

Keywords:
Kubernetes
Orchestration
Microservices
Reinforcement Learning
Resource allocation
Compute Continuum

ABSTRACT

With the advent of containerization technologies, microservices have revolutionized application deployment by converting old monolithic software into a group of loosely coupled containers, aiming to offer greater flexibility and improve operational efficiency. This transition made applications more complex, consisting of tens to hundreds of microservices. Designing effective orchestration mechanisms remains a crucial challenge, especially for emerging distributed cloud paradigms such as the Compute Continuum (CC). Orchestration across multiple clusters is still not extensively explored in the literature since most works consider single-cluster scenarios. In the CC scenario, the orchestrator must decide the optimal locations for each microservice, deciding whether instances are deployed altogether or placed across different clusters, significantly increasing orchestration complexity. This paper addresses orchestration in a containerized CC environment by studying a Reinforcement Learning (RL) approach for efficient microservice deployment in Kubernetes (K8s) clusters, a widely adopted container orchestration platform. This work demonstrates the effectiveness of RL in achieving near-optimal deployment schemes under dynamic conditions, where network latency and resource capacity fluctuate. We extensively evaluate a multi-objective reward function that aims to minimize overall latency, reduce deployment costs, and promote fair distribution of microservice instances, and we compare it against typical heuristic-based approaches. The results from an implemented OpenAI Gym framework, named as *HephaestusForge*, show that RL algorithms achieve minimal rejection rates (as low as 0.002%, 90x less than the baseline Karmada scheduler). Cost-aware strategies result in lower deployment costs (2.5 units), and latency-aware functions achieve lower latency (268–290 ms), improving by 1.5x and 1.3x, respectively, over the best-performing baselines. *HephaestusForge* is available in a public open-source repository, allowing researchers to validate their own placement algorithms. This study also highlights the adaptability of the DeepSets (DS) neural network in optimizing microservice placement across diverse multi-cluster setups without retraining. The DS neural network can handle inputs and outputs as arbitrarily sized sets, enabling the RL algorithm to learn a policy not bound to a fixed number of clusters.

1. Introduction

In recent years, the landscape of application deployment and life-cycle management has experienced a profound transformation with the advent of containers [1]. From traditional monolithic structures, applications have evolved into intricate loosely-coupled microservices, yielding substantial enhancements in deployment flexibility and operational efficiency [2]. Nevertheless, the efficient management of modern microservice-based applications demands sophisticated orchestration solutions. The rise of innovative paradigms such as Fog Computing [3,4], Edge Computing [5,6], and the Compute Continuum (CC) [7,8] puts even more pressure on popular cloud infrastructures to support novel use cases: highly-mobile self-driving vehicles [9], bandwidth-intensive Extended Reality (XR) applications, and ultra-reliable Industrial Internet of Things (IIoT) services [10]. These use cases require computing resources closer to devices and end-users, but the lack of efficient multi-cluster management features has hindered the deployment of these applications due to their stringent latency and bandwidth requirements [11].

The existing literature primarily focuses on single-cluster scenarios with a few works addressing multi-cluster orchestration [12,13]. The

* Corresponding author.
E-mail address: josepedro.pereiradossantos@ugent.be (J. Santos).

https://doi.org/10.1016/j.future.2024.107680
Received 2 July 2024; Received in revised form 6 December 2024; Accepted 14 December 2024
Available online 1 January 2025

orchestration dilemma becomes notably more complex in these multi-cluster scenarios: several clusters add a layer of complexity to the overall system due to their unique configurations, resource capacities, and network settings. Ensuring low latency across geographically distributed clusters poses a significant challenge, which is even exacerbated by the complexity of container-based applications, typically composed of multiple microservices, each with different and strict requirements. Therefore, orchestrator solutions for the CC must make decisions regarding the deployment location of each microservice, including whether to concentrate instances within a single cluster or distribute them across multiple clusters. This approach helps overcome the limitations of traditional scheduling methods in managing the dynamic changes inherent in microservice-based architectures. To achieve this, an effective strategy is essential for determining when to distribute microservice instances across different clusters focused on optimizing various performance factors such as deployment costs and the application's latency. Moreover, since individual microservices often exhibit varying workloads based on user demand and application complexity, efficient orchestration mechanisms are crucial to prevent compute node overload and ensure that computing resources are used efficiently within the CC.

This paper addresses these challenges by studying a Reinforcement Learning (RL)-based approach for microservice placement within the CC across a multi-cluster Kubernetes (K8s) infrastructure, a widely adopted container orchestration platform [14]. Specifically, this manuscript builds upon the foundations of [15], where we initially explored the adoption of RL techniques to solve the microservice multi-cluster deployment problem. In [15], we leveraged the capabilities of the open-source project Kubernetes Armada (Karmada) [16], a control-plane solution for managing multi-cluster applications in hybrid cloud environments, aiming to overcome the baseline heuristics implemented in Karmada. Motivated by the preliminary but promising results presented in [15], this work proposes a novel RL-based OpenAI Gym environment named *HephaestusForge*[1] to provide a scalable and cost-effective solution to train RL agents for this problem.

Instead of considering a single-objective reward model, such as the one presented in [15], *HephaestusForge* introduces a multi-objective reward function that considers three different performance factors: latency, cost, and inequality. This approach enables the learning of more effective deployment strategies capable to cope with the several challenges that characterize CC scenarios. The results based on the implemented OpenAI Gym environment show that RL can find efficient microservice placement schemes while prioritizing latency reduction, favoring low deployment costs, and avoiding distribution inequality compared to heuristic-based methods.

Furthermore, this work assesses the ability of DeepSets (DS) neural networks [17] to generate models that address the CC orchestration challenge across different setups without retraining. In DS neural networks, inputs and outputs are arbitrarily sized sets, meaning that the policy learned by the RL algorithm can be applied to diverse multi-cluster scenarios with a varying number of managed clusters. This is a remarkable capability that effectively addresses the dynamic nature of CC scenarios, where clusters, particularly edge clusters, can be added or removed frequently. The main contributions of the paper are threefold:

- ***HephaestusForge* framework**: This work extends the *gym-multi-k8s* environment presented in [15] by considering a multi-objective reward function that addresses different performance factors (details in Section 4) and refining the RL-based approach for proper scheduling of microservice applications within the CC. *HephaestusForge* has also been open-sourced,[2] enabling researchers to leverage the framework to evaluate their orchestration ideas.
- **Extensive Evaluation**: *HephaestusForge* has been validated in numerous multi-cluster K8s setups that considered different RL algorithms and several heuristic-based methods under dynamic conditions, where latency and resource allocation (CPU and memory) vary depending on the chosen orchestration actions (Section 5). Results show that *HephaestusForge* can find near-optimal allocation schemes for the selected strategy while achieving higher performance than typical heuristics.
- **RL scalability and generalization**: This work also evaluates the generalization potential of the DS neural network by applying it to different problem sizes without retraining. Results show that DS algorithms can effectively optimize microservice placement in larger multi-cluster scenarios. Additionally, regarding scalability, DS algorithms achieve significantly lower rejection rates compared to common heuristics, even as orchestration complexity increases with the number of microservice replicas requiring deployment (Section 6).

The remainder of the paper is organized as follows: Section 2 discusses the literature on multi-cluster orchestration. Section 3 highlights the importance of efficient multi-cluster orchestration while describing the envisioned end-to-end system based on its integration with the Karmada open-source project. Section 4 details the RL-based approach, including its observation states, action spaces, and the multi-objective reward function. Section 5 describes the evaluation setup, followed by the results in Section 6. Finally, Section 7 focuses on open challenges and future directions, and Section 8 concludes this paper.

2. Related work

Orchestration in Cloud Computing has been an active research topic in recent years [31], with numerous studies proposing scheduling strategies to enhance container allocation within leading orchestration platforms. This literature section reviews relevant works concerning application deployment, particularly emphasizing recent orchestration techniques tailored for multi-cluster infrastructures within the CC. The awareness of the orchestrator plays a major role in these scenarios since it will allow more refined scheduling decisions aimed at maintaining system performance and responsiveness.

Heuristics and Theoretical Models have been vastly studied in the literature [18–20,26,27]. For instance, in [20], the authors introduce three scheduling algorithms tailored for heterogeneous cloud environments. These algorithms aim to identify the most suitable task locations while optimizing metrics such as makespan, resource utilization, and throughput. Similarly, in [26], the authors present a multi-objective algorithm based on reliability considerations. Their results show the effectiveness of the approach compared to other analogous algorithms in solving multi-objective workflow scheduling problems in multi-cloud systems. However, a notable limitation of these methodologies lies in their platform-specific design, reducing their potential applicability in operational environments [32,33]. In addition, theoretical formulations such as Integer Linear Programming (ILP) and Mixed-Integer Linear Programming (MILP) models suffer from high execution time in finding the optimal placement scheme which hinders their applicability in real platforms [34]. Heuristics typically perform well for simple objectives, but these may encounter limitations with increased complexity, especially when multiple performance factors come into play.

Scheduling strategies for the K8s platform have received significant attention in the last few years [16,21,22,24,29,30]. Most efforts

[1] *HephaestusForge* is inspired by the Greek god Hephaestus, who is associated with craftsmanship, technology, and innovation. The addition of *Forge* emphasizes the idea of creation and refinement, reflecting the complex process of efficiently deploying microservices in Kubernetes multi-clusters.

[2] https://github.com/jpedro1992/HephaestusForge

Table 1
Comparison of existing works related to multi-cluster application deployment.

Authors	Year	Virtualization	Dimension	Main Focus	Generalization	Evaluation Method
Bhamare, D. et al. [18]	2017	VMs	N	R & NL	×	S
Guerrero, C. et al. [19]	2018	VMs & C	MO	R & NL	×	S
Panda, S. K. et al. [20]	2019	VMs	MO	R & M	×	S
Lee, S et al. [21]	2020	C	R	R	✓	K8s
Rossi, F, et al. [22]	2020	C	N	NL	✓	K8s
Karmada [16]	2020	C	L	R	✓	K8s
Zhang, Y. et al. [23]	2020	N/A	R	R & NL	×	S
Tamiru, M. A. et al. [24]	2021	C	R	R	✓	K8s
Shi, T. et al. [25]	2021	VMs	L	R & NL	×	S
Qin, S. et al. [26]	2023	VMs	MO	RL	×	S
Moreno-V., R. et al. [27]	2024	VMs & C	MO	R & NL	×	S
Suzuki, A. et al. [28]	2023	N/A	N	R & NL	×	S
Santos, J. et al. [15]	2024	C	N + R	R & NL	✓	RL
Zaccarini, M. et al. [29]	2024	N/A	MO	AR & L	×	S
Ejaz, S. et al. [30]	2024	C	N + L	NL & R	×	K8s
HephaestusForge	2024	C	MO	R & NL & DI	✓	RL

Virtualization: VMs = Virtual Machines, C = Containers, N/A = no clear distinction.
Dimension: N = Network-aware, MO = Multi-objective, L = Location-aware, R = Resource-aware.
Main Focus: AR = Acceptance of Requests, R = Resources, RL = Reliability, NL = Network Latency, M = Makespan, DI = Distribution Inequality.
Generalization: ✓= addressed, ×= not considered.
Evaluation Method: K8s = Kubernetes, S = Simulation, RL = RL environment.

aim to improve resource efficiency [21,24], satisfy deployment requests [29], or reduce the application's response time by considering the network latency between geographically distributed clusters [22,30], showing the benefits of network-aware placement. Karmada [16] scheduling focuses on deployment preferences specified by cloud administrators. Microservice replicas can be deployed within a single cluster or distributed across multiple clusters. If the spreading policy is chosen, a simplified cluster resource modeling approach determines how replicas are spread across clusters. Additionally, in [30], the authors propose a novel solution for orchestrating fog-native workflows over geo-distributed clusters, possibly from different domains. The approach decomposes a workflow into a set of cluster-specific dependency sub-graphs. Results have shown that distributing CPU-light microservices introduces a significant network overhead. However, similar to Karmada, the distribution of replicas is based on the application administrator's preferences. In contrast, *HephaestusForge* aims to find the optimal decision based on real-time infrastructure status, alleviating the need for administrators to specify replica deployment preferences across the entire CC.

RL-based orchestration approaches have emerged as promising alternatives to conventional heuristics in recent years [15,23,25,28]. These methods aim to train RL algorithms on how to efficiently deploy microservices in a multi-cluster environment by providing real-time infrastructure status updates following each action. RL approaches exhibit robustness towards dynamic demands, thanks to their ability to adapt model parameters in response to significant events (i.e., online learning). However, the primary drawback of RL techniques lies in the long execution time required to converge to a stable model, potentially resulting in inefficient scheduling actions during the learning phase. Previous studies [35,36] have shown that training RL agents in operational environments offers higher reliability. However, it is significantly more expensive, on average 100 times more time-consuming than simulation-based environments. In fact, [36] revealed that RL agents trained in simulation achieved comparable performance to those trained in cluster environments, highlighting the importance of offline training in delivering similar results at a fraction of the cost. Thus, the approach proposed in this paper offers a more scalable and cost-effective solution for RL training through the use of the *HephaestusForge* framework, a near-real simulation-based environment.

Table 1 provides a comparative analysis of the discussed works in chronological order, categorizing them based on their primary characteristics. However, conducting a quantitative assessment poses challenges due to the specialized nature of these techniques tailored for specific systems or virtualization technologies. To the best of our knowledge, no standardized testing framework for multi-cluster scheduling exists. Leveraging insights from our previous study named *gym-multi-k8s* [15], *HephaestusForge* extends the previous framework by considering a multi-objective reward function that addresses different performance factors such as deployment cost, expected latency, and distribution inequality during microservice deployment. In addition, this paper evaluates the RL generalization potential of the DS neural network. The objective is to train an RL agent in a small-scale scenario and subsequently apply the learned policy in large-scale multi-cluster infrastructures.

3. Towards efficient orchestration within the Compute Continuum (CC)

This section starts by discussing how microservices can be deployed across several K8s clusters through the use of existing tools, while highlighting our envisioned RL-based approach in the CC scenario. Lastly, the Karmada scheduling algorithm is detailed.

3.1. Microservice deployment in Kubernetes multi-clusters

K8s offers robust solutions for microservice deployment across several clusters, providing dynamic and adaptive mechanisms to distribute microservice instances. The most relevant approaches today include:

- **Kubernetes Federation (KubeFed)**[3] (archived and no longer active since April 2023) was one of the first official K8s projects focused on multi-cluster scenarios designed to manage multiple K8s clusters as a single entity. It allowed for centralized control and configuration management across clusters. KubeFed provided features for multi-cluster application deployment, scaling, and monitoring, making it suitable for scenarios requiring centralized management and control.

[3] https://github.com/kubernetes-retired/kubefed

- **Cluster API**[4] is a K8s subproject initiated by the Kubernetes Special Interest Group (SIG) Cluster Lifecycle dedicated to streamlining the provisioning, upgrading, and management of multiple K8s clusters through declarative APIs and tooling. Leveraging K8s-style APIs and patterns, the Cluster API project automates cluster lifecycle management for platform operators. It supports numerous infrastructure components such as Virtual Machines (VMs) networks, and load balancers in a consistent manner ensuring reproducible cluster deployments across diverse infrastructure environments.
- **Open Cluster Management (OCM)**[5] is a community-driven project focused on multi-cluster and multi-cloud scenarios for K8s apps. Open APIs are evolving within this project for cluster registration, workload distribution, and the dynamic placement of policies and workloads.
- **Karmada**[6] is a K8s management system that enables to run cloud-native applications across multiple K8s clusters and clouds, with no modifications needed in the applications. By supporting K8s-native APIs and providing advanced scheduling capabilities, Karmada enables an open hybrid cloud K8s environment, with key features such as centralized multi-cloud management, high availability, failure recovery, and traffic scheduling.

This work envisions a multi-cluster scenario within the CC as an aggregation of multiple heterogeneous K8s clusters managed singularly by a control-plane entity. It will enable a dynamic methodology for developing, deploying, and managing all the distributed computing layers in the CC (i.e., edge, fog, and cloud). The proposed approach uses K8s at every layer of the CC due to its various heterogeneous distributions such as MicroK8s, Kubedge, and K3s [37]. With a large variety of managed K8s setups, it is logical to consider this scenario as a federation of multi-cluster environments. Therefore, Karmada has been adopted as a federation layer capable of deploying cloud-native applications across multiple K8s clusters as shown in Fig. 1. The main reason to choose Karmada as our federation layer is that it seems a more mature solution than the others analyzed in this section, such as OCM. For instance, Karmada can already exploit the K8s Native Application Programming Interface (API) in the resource templates, making it easier to integrate with the plethora of existing K8s tools and extend it with plugins.

Application and cluster administrators can submit deployment files (typically YAML files) to the cluster to instantiate *Services* and *Pods*. K8s introduced the concept of a *Service* as an abstraction that defines a logical set of *Pods*, the smallest working unit in K8s that can host one or more containers, as shown in Fig. 1. The *Service* abstraction provides a *Service Load Balancer* that balances the incoming traffic across the associated *Pods*. This abstraction simplifies the process of load balancing for microservices, by hiding unnecessary complexity.

3.2. End-to-end system overview

In our OpenAI Gym-based RL scheduling system [38] named as *HephaestusForge*, several RL algorithms are available for training to generate an orchestration strategy using as input static and dynamic information about the K8s cluster (detailed further in Section 4). For example, allocated CPU and memory amounts in each cluster vary dynamically based on the number of deployed microservice replicas in each cluster. Also, the latency of each cluster varies dynamically as if network measurements are periodically executed in the cluster, similar to our recent network-aware scheduling approach presented in [39]. This information serves as input and is updated periodically after each action selected by the RL algorithm. The *HephaestusForge* framework has been developed to replicate the behavior of deployment requests via Karmada, providing the RL agent with relevant information available within a typical K8s cluster. By emulating the behavior of the Karmada scheduler, trained RL orchestration policies could then be later validated in operational environments by retrieving real-time information from the K8s cluster via popular monitoring platforms such as Prometheus [40]. However, this validation is left out of the scope of this paper, but planned as future work. Instead, in this work, we focus on the validation of the performance of the proposed multi-objective reward function in the *HephaestusForge* framework.

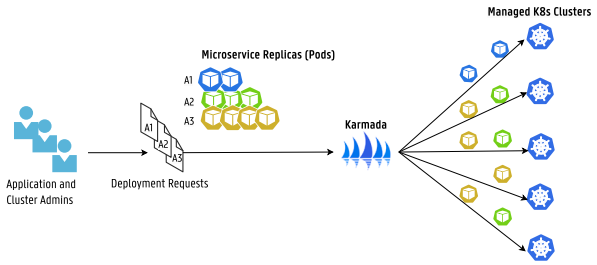


Fig. 1. Illustration of Microservice deployments via Karmada in a K8s multi-cluster infrastructure.

Table 2
The Karmada Scheduling algorithm based on Cluster Resource modeling. An example based on CPU capacities and pod CPU requirements.

	Available Clusters		
CPU Resources	Cluster 1	Cluster 2	Cluster 3
Capacity	4.0	4.0	2.0
Allocated	0.95	2.0	1.0
Pod's Requirements	**Microservice Replica Calculation**		
500 m (CPU)	$\frac{4-0.95}{0.5} = 6.1$	$\frac{4-2}{0.5} = 4$	$\frac{2-1}{0.5} = 2$
Cluster Selection	✓	×	×

3.3. Karmada scheduling algorithm

Despite the several functionalities available in Karmada, plenty of room exists for improvement in its standard behavior, especially regarding application scheduling. Karmada supports two modes for deploying microservice replicas in a K8s cluster: *duplicated* and *divided*. The first mode implies deploying the number of requested instances in all clusters, and the second strategy splits the number of requested replicas across all the clusters. Depending on the strategy favored by the cloud administrator, extra options (e.g., *ClusterAffinities*, *LabelSelectors*) can be inserted into the *PropagationPolicy* object to fine-tune the behavior of the Karmada scheduler. Karmada decides to divide replicas mainly by the resource availability of each cluster,[7] including CPU and memory usage, and pod quotas. It utilizes cluster resource modeling to assess the free and allocated resources of each cluster, thus enabling informed scheduling decisions.

For example, when evaluating candidate clusters for a specific pod, Karmada considers not only the absolute resource capacity but also the current resource utilization, ensuring efficient resource allocation. Karmada favors scheduling on clusters with excess resources based on the pod's deployment requirements, aiming to enhance fault tolerance and scalability, as it mitigates the risk of resource exhaustion and facilitates efficient replica distribution across the multi-cluster federation. Table 2 presents an example in which the pod's CPU request is 0.5 CPU (500 millicpu). All clusters have sufficient CPU resources to run the

[4] https://github.com/kubernetes-sigs/cluster-api
[5] https://open-cluster-management.io/
[6] https://karmada.io/
[7] https://karmada.io/docs/userguide/scheduling/cluster-resources

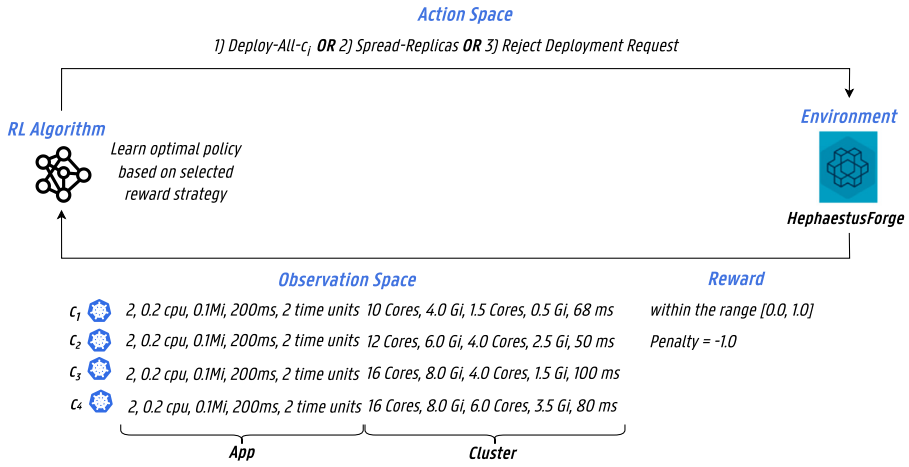


Fig. 2. Overview of the *HephaestusForge* framework.

pod, but the Karmada scheduler prefers to schedule the pod to cluster 1 based on its available and allocated CPU resources. Though, Karmada does not consider the current latency of the clusters while scheduling. Also, the duplication policy typically leads to resource wastage since the demand is lower than the number of reserved resources.

The proposed RL-based approach aims to automate microservice deployment within the CC in multi-cluster scenarios by finding an optimal balance between deploying the number of requested replicas in a single cluster or distributing them across several ones. A multi-objective reward function has been developed focused on resource efficiency, network latency, and distribution inequality to find a suitable multi-cluster placement for different application scenarios.

4. Reinforcement Learning (RL)-based multi-cluster orchestration

4.1. HephaestusForge framework

An OpenAI Gym-based environment [38] named *HephaestusForge* has been developed to train the RL algorithms in a scalable and cost-effective manner, as illustrated in Fig. 2. The framework enables RL agents to learn how to deploy microservices in multi-cluster scenarios efficiently. The environment consists of a discrete-event RL scenario to reenact the behavior of multiple deployment requests for a given microservice deployed via Karmada on several K8s clusters. During training, the amount of resources available in each cluster is updated based on the actions chosen by the RL agent. Section 5 shows the deployment requirements used for the RL environment based on a realistic microservice-based application to create near-real experiments.

In addition, the proposed approach adopts the DS methodology presented in [15,41]. Deep RL methods based on Multi-Layer Perceptrons (MLPs) operate in fixed-length vector spaces, which cannot support variable input or output dimensionalities. In other words, for the microservice scheduling problem, if an MLP-based RL agent learns on a multi-cluster setup with four clusters, it cannot be directly applied to a different multi-cluster scenario that manages eight clusters. Instead, the DS neural network assumes that inputs and outputs can be arbitrarily-sized sets. Therefore, the learned policy by the RL agent is not bound to a fixed number of clusters, and it can generalize its learned policy to different multi-cluster scenarios without retraining. The use of the DS neural network allows the proposed *HephaestusForge* to generalize well to problems with larger sizes than the ones seen during training. This is a remarkable capability that well suits CC scenarios, which often present variability of available computing resources, e.g., edge nodes stop working or additional computing resources are added to clusters.

4.2. Observation space

Table 3 shows the observation space considered for the multi-cluster orchestration problem, describing the environment at a given step. It includes two sets of metrics: *App* and *Cluster*. The first set *App* corresponds to the deployment requirements of the microservice-based application, including the number of requested replicas that need to be deployed (R), the CPU and memory requests for each replica (ω_{cpu} and ω_{mem}), the latency threshold (in milliseconds) that the cluster or clusters hosting the required replicas should respect (Δ_r), and the expected inter-arrival time (T_r) of the request r measured in time units, representing the time interval between successive microservice requests. Since the RL agent is invoked with each service request arrival, the time interval between consecutive steps can vary. To help the RL agent learn the environment dynamics and to capture changes in resource consumption across clusters between successive observation states, T_r is included in the observation space.

The second set *Cluster* corresponds to the metrics related to the current status of the infrastructure, such as resource capacities (Π_{cpu} and Π_{mem}), allocated amounts of resources (Θ_{cpu} and Θ_{mem}), and the current latency of the cluster, which is calculated based on several latency metrics depending on the number of available clusters in the multi-cluster scenario. The allocation of CPU and memory resources for each cluster is directly influenced by the number of hosted replicas. As the number of deployed replicas grows, the allocated CPU and memory increase proportionally to the specified requirements for each replica. Similarly, when a deployment is terminated, the allocated resources are adjusted to reflect the freed CPU and memory. The available free CPU (Ω_{cpu}) and memory (Ω_{mem}) resources are given by Eqs. (1) and (2), respectively. However, these metrics are excluded from the observation space since their inclusion did not enhance the performance of the algorithms in our experiments since the agents already possess knowledge of the total capacity and the allocated amount of resources. Each cluster in our system is characterized by an average latency, which reflects the communication latency from cluster c to all other available clusters. These latency values are randomly initialized between 1 and 1000 ms. Thus, different patterns occur in consecutive episodes to analyze the RL agent's generalization capabilities.

Also, Table 4 shows detailed resource capacities for each cluster based on different cluster types and their corresponding deployment cost. Resource capacities are then represented as values in $[2.0, 32.0]$, and allocated resources are initiated as values in $[0.0, 0.2]$ since each cluster has a reserved amount of resources for background services (e.g., monitoring). This information helps the agent to select adequate actions at a given moment from the action space described next.

$$\underbrace{\Omega_{cpu}}_{\text{free CPU}} = \underbrace{\Pi_{cpu}}_{\text{CPU Capacity}} - \underbrace{\Theta_{cpu}}_{\text{CPU Allocated}} \tag{1}$$

Table 3
The structure of the Observation Space.

Set	Metric	Description
App	R	The number of requested replicas.
	ω_{cpu}	The CPU request of each replica (in millicpu).
	ω_{mem}	The memory request of the replica (in mebibyte).
	Δ_r	The latency threshold of the request (in ms).
	T_r	The inter-arrival time of the request (in time units).
Cluster	Π_{cpu}	The cluster's cpu capacity.
	Π_{mem}	The cluster's memory capacity.
	Θ_{cpu}	The CPU allocated in the cluster.
	Θ_{mem}	The memory allocated in the cluster.
	δ_c	The average latency of cluster c. It is calculated based on the communication latency from cluster c to all available clusters.

Table 4
The hardware configuration of each cluster based on Amazon EC2 On-Demand Pricing [42].

Cluster Type	Amazon Cost ($/h)	Cost (τ_c)	CPU	RAM
Cloud	2xlarge (0.2688)	16.0	8.0	32.0
Fog Tier 2	xlarge (0.1344)	8.0	4.0	16.0
Fog Tier 1	large (0.0672)	4.0	2.0	8.0
Edge Tier 2	medium (0.0336)	2.0	2.0	4.0
Edge Tier 1	small (0.0168)	1.0	2.0	2.0

$$\underbrace{\Omega_{mem}}_{\text{free Memory}} = \underbrace{\Pi_{mem}}_{\text{Memory Capacity}} - \underbrace{\Theta_{mem}}_{\text{Memory Allocated}} \tag{2}$$

4.3. Action space

Table 5 shows the action space designed for *HephaestusForge* as a discrete set of possible actions, where a single action is applied at each timestep. Given a deployment request, the RL agent decides to either allocate the total number of replicas to a single cluster, divide the number of instances across all available clusters, or reject the request. The aim of the *spread* policy is to consider all available clusters for deployment, ensuring that at least two clusters are used to deploy all replicas of a request. The total size of the action space depends on the total number of clusters in the CC scenario. Let us assume that the multi-cluster setup consists of C clusters, the action space length is then $C+2$. Rejection is allowed since computational resources might be scarce at a given moment, and no cluster can satisfy the request. The agent should not be penalized in these cases. Regarding penalties (i.e., negative reward), a simple approach commonly followed in the literature [43] is to penalize the agent if it selects an invalid action since these are typically known beforehand based on the allocated computing resources. In contrast, action masking [44] can teach the agent that depending on the current state s specific actions are invalid. This approach has recently shown significantly higher performance and sample efficiency than penalties. The action masks for each cluster c in state s can be defined as follows:

$$mask(s)[c] = \begin{cases} true & \text{If cluster } c \text{ has enough resources.} \\ false & \text{Otherwise.} \end{cases} \tag{3}$$

Whereas for spread and reject actions, the action mask is always *true*, avoiding the lock in case all actions are marked invalid. It is noteworthy that the current Karmada does not make this decision between *deploy-all* and *spread* placement. The cloud administrator decides by indicating the preferred strategy in the deployment file. *HephaestusForge* aims to find the optimal balance between both policies by following a First Fit Decreasing (FFD) approach for the *spread* action (Alg. 1). This balance is affected by the selected reward function. The FFD heuristic sorts clusters in descending order based on their capacity in terms of computing resources. For each cluster, a minimum factor (f) is calculated based on the minimum ratio of available CPU and memory to the replica's requested CPU and memory. This determines how many replicas can fit in each cluster based on the more limiting resource (CPU or memory). A threshold (Δ) is updated to the minimum value of f across all clusters, ensuring Δ reflects the most constrained cluster's ability to accommodate replicas. Then, the algorithm iterates over the sorted clusters, placing replicas while respecting both CPU and memory constraints. If replicas remain to be deployed after the first distribution loop (*DistLoop*), the algorithm repeats the loop until all replicas are placed. Regarding the sorting criteria, several FFD heuristics have been tested by the authors, including those based on free CPU, free memory, and a combination of both with equal weighting. After analyzing the results, prioritizing free CPU yielded higher average rewards for the RL agent. Thus, free CPU has been selected as the sorting function in the

Algorithm 1 First Fit Decreasing (FFD)

Input: R, the number of requested replicas. C, the number of clusters. $\omega_{cpu,mem}$, the replica's requested cpu/memory. $\Omega_{cpu,mem}$, the cluster's amount of free cpu/memory.
Output: α, the distribution of replicas across all clusters

```
if R = 1 then
    penalty ← true                                  ▷ Penalize the agent
    return α = 0
end if
min ← 1, max ← R, Δ ← R                             ▷ Get min and max replicas
for each c ∈ C do                                   ▷ Calculate min factor
    f ← min(Ω_cpu[c]/ω_cpu[c], Ω_mem[c]/ω_mem[c])
    Δ ← min(f, Δ)
end for
if Δ ≥ R then
    Δ ← R − 1                                       ▷ To really distribute replicas
end if
S ← sorted(Ω_cpu)                                   ▷ Sort by decreasing order of CPU
for each c ∈ S do                                   ▷ DistLoop: distribute replicas
    if R = 0 then
        break
    else if R > 0 ς Δ < R ς (ω_cpu × Δ < Ω_cpu[c]) ς (ω_mem × Δ < Ω_mem[c]) then
        α[c] = α[c] + Δ
        R = R − Δ
    else if R > 0 ς (ω_cpu < Ω_cpu[c]) ς (ω_mem < Ω_mem[c]) then
        α[c] = α[c] + min
        R = R − min
    end if
end for
if R = 0 then
    return α
else if R ≠ 0 then
    repeat
        DistLoop                                    ▷ Repeat the DistLoop
    until R = 0
end if
```

Table 5
The structure of the Action Space.

Action Name	Description
Deploy-all-c	Deploy all replicas in cluster c.
Spread	Divide and spread replicas across different clusters.
Reject	The agent rejects the request. Nothing is deployed.

FFD implementation. FFD is widely used for its simplicity and effectiveness, often yielding near-optimal solutions with a time complexity of $O(c \log c)$, where c is the number of clusters. This study considers a multi-objective reward function with three performance factors, as described next.

4.4. Reward function

The purpose of a reward function is to guide the RL agent towards maximization of accumulated rewards by choosing appropriate actions depending on the observation state. A multi-objective reward function (4) has been designed based on three different objectives: cost-aware (5), latency-aware (6), and inequality-aware (7). If the agent accepts the deployment request, it receives a positive reward based on these strategies and its corresponding weights (ω_l, ω_c and ω_i), normalized between $[0.0, 1.0]$. Otherwise, the agent is penalized if it decides to reject the request (i.e., -1), and computing resources were available to deploy all requested microservice replicas.

$$r = \begin{cases} \omega_c \times r_{cost} + \omega_l \times r_l + \omega_i \times r_{ineq} & \text{if req. is accepted.} \\ -1 & \text{if req. is rejected.} \end{cases} \tag{4}$$

$$r_{cost} = 1.0 - \Gamma_d \quad \text{where:} \quad \Gamma_d = \text{Expected Deployment Cost for all replicas} \tag{5}$$

$$r_{latency} = 1.0 - \lambda_d \quad \text{where:} \quad \lambda_d = \text{Expected Latency for deployment request} \tag{6}$$

$$r_{inequality} = 1.0 - G \quad \text{where :} \quad G = Gini\ Coefficient \tag{7}$$

The **cost-aware** function leads the agent to deploy replicas on clusters focused on minimizing the allocation cost (i.e., τ_c). Cloud-type clusters are considerably more expensive than fog and edge types, often providing higher latency. Therefore, the agent favors deploying microservice replicas to edge or fog clusters due to the higher rewards associated with these actions. However, fog and edge clusters typically possess fewer computing resources than cloud nodes, meaning these cluster types might not be able to host all required replicas for a given deployment request. The deployment cost of a request, denoted by Γ_d, is calculated as the average of the deployment costs for all replicas within the request as follows:

$$\Gamma_d = \frac{1}{R} \sum_{i=1}^{R} \tau_{c_i} \tag{8}$$

where:

Γ_d is the deployment cost of the microservice request.

R is the number of replicas within the request.

τ_{c_i} represents the allocation cost of the ith replica.

Each replica's cost is proportional to the allocation cost (τ_c), which varies based on the cluster type. When the RL agent selects the spread action, microservice replicas can incur different deployment costs depending on the clusters to which they are assigned. The rationale behind the deployment cost formulation is to enable the RL agent to learn a strategy that minimizes overall deployment costs by balancing replicas across different cluster types. In contrast, a cost formulation based on the summation of individual replica costs would disproportionately favor deployments with fewer replicas, potentially skewing the agent's strategy. Using the average cost provides a normalized metric that ensures consistency in decision-making across different replicas and cluster types.

The **latency-aware** function aims to minimize the expected latency of the deployment request. The latency of a microservice request is influenced by whether all replicas are deployed within the same cluster or distributed across multiple clusters since each individual replica contributes to the overall average latency of the microservice request. Consequently, the expected latency (λ_d) is calculated based on the average latency of each replica within the deployment request, considering the specific clusters where these replicas are hosted:

$$\lambda_d = \frac{1}{R} \sum_{i=1}^{R} \delta_{c_i} \tag{9}$$

where:

λ_d is the expected latency of the microservice request.

R is the number of replicas within the request.

δ_{c_i} represents the latency of the ith replica.

The adoption of an average latency instead of considering multiple individual latencies lies in the need to improve convergence speed and ease the training process of the RL agent. Specifically, the increased dimensionality of the state space and the need for the agent to track and optimize multiple individual latencies rather than a single average can significantly slow down the learning process. Although we acknowledge that this design choice could lead to less granular performance, we favored operating with a smoother decision-making process. The study of more complex state representations will be addressed in future work. In addition, using the average latency simplifies the model, and encourages the RL agent to minimize communication delays more equitably, avoiding excessive penalization of any single replica's latency. Future work will explore alternative formulations, particularly for scenarios where communication bottlenecks or strict latency thresholds are critical considerations.

Lastly, the **inequality-aware** function leads the RL agent to choose deployment actions that evenly distribute replicas across the number of available clusters. The reward is calculated based on the *Gini Coefficient* (G) [45] that ranges from $[0.0, 1.0]$, where 0 means perfect equality (all clusters host the exact number of replicas), and 1 indicates perfect inequality (all replicas deployed in one cluster). A lower *Gini Coefficient* indicates then a more equitable distribution. The *Gini Coefficient* is an accurate measure of inequality in a distribution, calculated using the formula:

$$G = \frac{\sum_{i=1}^{c} \sum_{j=1}^{c} |L_i - L_j|}{2c^2 \bar{L}} \tag{10}$$

where:

G is the Gini coefficient.

c is the number of clusters.

L_i is the number of replicas deployed by cluster i.

$\bar{L}$ is the average number of replicas across all clusters.

4.5. The DeepSets neural network

DS [17,41] is a neural network architecture specifically designed to process sets as inputs, addressing the challenges associated with their unordered and variable-sized nature. Traditional neural networks, which typically expect inputs to be ordered and of fixed size, are not well-suited for handling sets. In the DS methodology [41], the input data to our neural network is represented as a set. Formally, let the input be a set of c elements, $X = \{x_1, \ldots, x_c\}$. The aim is to design neural networks that are either Permutation-Invariant (PI) or Permutation-Equivariant (PE) concerning the ordering of elements in this set. A function f is **PI** if $f(X) = f(p(X))$ for any permutation p, meaning the output of the function is independent of the ordering of the

input set elements. In contrast, a function f is **PE** if $f(p(X)) = p(f(X))$ for any permutation p, implying that permutations of the input are reflected in the output.

In this study, DS are applied to implement neural networks that are either PI or PE. DS-based RL algorithms offer several advantages towards designing scalable orchestrators for the CC:

1. **Permutation Handling:** DS can be designed to be either PI or PE, meaning the learned functions are not constrained by a specific indexing of the number of clusters.
2. **Scalability:** The time complexity of DS scales linearly with the number of clusters, and computations can be easily parallelized over different elements in the set.
3. **Flexibility:** DS can process sets with an arbitrary number of elements, allowing generalization to varying numbers of clusters.

Formally, given sets of c elements, each characterized by m features, DS realize a permutation-equivariant function $f(x) : \mathbb{R}^{c\times m} \to \mathbb{R}^{c\times k}$ as follows:

$$f(x) = \sigma\left(x\Lambda - \mathbf{1}^T \Gamma \text{maxpool}(x)\right)$$

where $x \in \mathbb{R}^{c\times m}$ are the input features, Λ and $\Gamma \in \mathbb{R}^{m\times k}$ are trainable parameters, and $\sigma(\cdot)$ denotes an element-wise nonlinearity (e.g., ReLU). This formulation achieves PE with respect to the rows of x. The product $x\Lambda$ applies the same linear transformation Λ to each row of x, and the maxpool function extracts the maximum value from each column of x, ensuring permutation invariance [41]. Stacking multiple layers of this form builds PE neural networks, in which the computational flow is shown in Fig. 3(a).

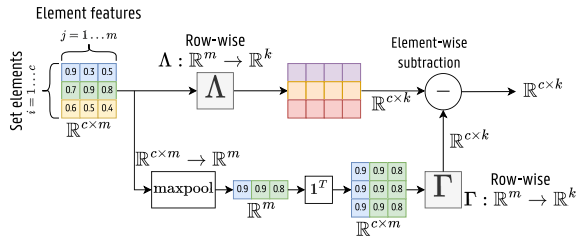
The operations described in the equation are independent of c, meaning that matrix multiplications can handle input sets of varying sizes. Once trained, DS networks can perform inference on sets of arbitrary size, with computational complexity scaling linearly with the number of elements in the set and parallelizable over the set elements, similar to batching in deep neural networks. In the context of actor-critic RL algorithms, PE DS can be utilized to implement a policy $\pi_\theta(a|s) : \mathbb{R}^{c\times m} \to \mathbb{R}^c$ for a discrete action space over the set elements. Here, the state $s \in \mathbb{R}^{c\times m}$ represents a set of c elements, each with m features, and the action $a \in \mathbb{R}^c$ denotes a categorical distribution over the c elements. Similarly, PI DS can be used to approximate the value function $V_\xi(s) : \mathbb{R}^{c\times m} \to \mathbb{R}$, where permutation invariance is necessary because the output is a scalar. To achieve permutation-invariance, we apply a PI pooling operator pool $: \mathbb{R}^{c\times m} \to \mathbb{R}^m$ (e.g., max, sum, or mean pooling) to the final layer of an equivariant DS network, producing a fixed-size representation of the input set [41]. This representation can then be transformed by a function $f : \mathbb{R}^m \to \mathbb{R}$ (e.g., a small MLP) to yield the desired scalar output. Fig. 3(b) illustrates the computational flow of PI DS.

5. Evaluation setup

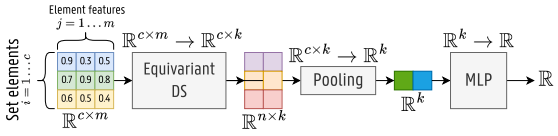
This section presents an overview of the several reward strategies used to validate the *HephaestusForge* framework by analyzing the Cloud2Edge (C2E) application. Section 5.1 details the reward strategies applied in the evaluation, followed by Section 5.2 which describes the evaluated DS-based RL algorithms. Lastly, Section 5.3 describes C2E and shows its deployment requirements, and Section 5.4 details the dynamics of the *HephaestusForge* framework.

5.1. Reward strategies

Table 6 details eight distinct reward strategies considered in the evaluation of the *HephaestusForge* framework:



(a) Permutation-Equivariant (PE).



(b) Permutation-Invariant (PI).

Fig. 3. Computational flow of Permutation-Equivariant (PE) and Permutation-Invariant (PI) DeepSets (DS) neural networks [41].

- ***Latency***: prioritizes minimizing latency above all other factors, assigning the highest weight ($\omega_l = 1.0$) to latency reduction. The RL agent focuses solely on reducing the expected latency of the deployment request, disregarding cost and inequality considerations.
- ***Cost***: aims to minimize deployment costs, with ω_c set to 1.0, without explicit considerations regarding latency or inequality.
- ***Inequality***: focuses on addressing inequality among clusters, with ω_i set to 1.0. The RL agent seeks to balance the deployment of replicas across clusters to mitigate disparities and ensure fair utilization.
- ***LatCost***: strikes a balance between minimizing latency and deployment costs, assigning equal weight ($\omega_l = \omega_c = 0.5$) to both objectives. The RL agent aims to achieve a compromise between latency reduction and cost efficiency in its deployment decisions.

- ***LatIneq***: balances latency reduction and inequality considerations, with $\omega_l = \omega_i = 0.5$. This strategy aims to minimize latency, while also addressing fairness across clusters.
- ***CostIneq***: prioritizes minimizing deployment costs while also addressing inequality, with $\omega_c = \omega_i = 0.5$. The goal is to achieve cost-efficient deployments while ensuring fair allocation across clusters.
- ***Balanced***: seeks a proportional approach, with moderate weights assigned to each objective ($\omega_l = 0.4$, $\omega_c = 0.3$, $\omega_i = 0.3$). The RL agent aims to achieve a compromise that considers all three factors in its deployment decisions.
- ***FavorLat***: emphasizes minimizing the latency by assigning a higher weight ($\omega_l = 0.6$) compared to cost and inequality considerations ($\omega_c = \omega_i = 0.2$).

By evaluating numerous reward strategies, the *HephaestusForge* framework aims to provide insights into the effectiveness of different allocation strategies in the deployment of microservices across distributed computing environments.

5.2. RL algorithms

Two DS-based RL algorithms that support discrete action spaces have been implemented based on the stable baselines 3 and CleanRL [46,47] libraries, both reliable implementations of RL algorithms written in Python. The RL algorithms have been adapted to use the DS neural network architecture by modifying their standard implementations in these popular RL libraries. The evaluated algorithms are the following:

Table 6
The evaluated reward strategies.

Name	ω_l	ω_c	ω_i
Latency	1.0	0.0	0.0
Cost	0.0	1.0	0.0
Inequality	0.0	0.0	1.0
LatCost	0.5	0.5	0.0
LatIneq	0.5	0.0	0.5
CostIneq	0.0	0.5	0.5
Balanced	0.4	0.3	0.3
FavorLat	0.6	0.2	0.2

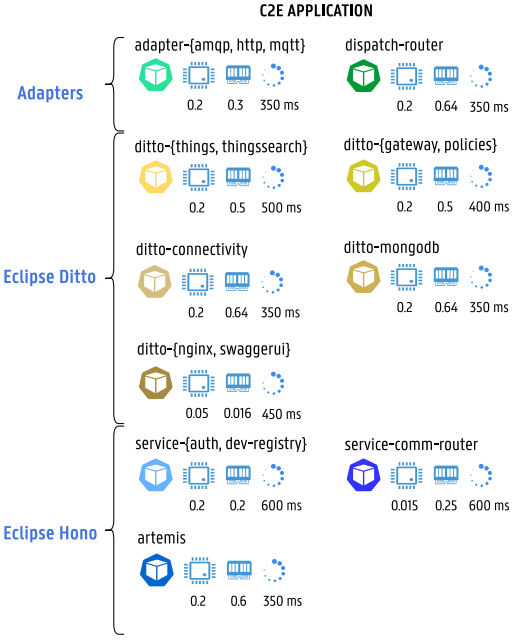


Fig. 4. Cloud2Edge (C2E) architecture and deployment requirements considered in the *HephaestusForge* evaluation.

- **DeepSets (DS)-Proximal Policy Optimization (PPO)**: PPO is a policy gradient method for RL vastly used today for different scenarios (e.g., robot control and video games). DS-PPO is similar to PPO but uses DS as its neural network. Also, is supports action masking.
- **DeepSets (DS)-Deep Q-Network (DQN)**: DQN combines the classical Q-Learning RL algorithm with deep neural networks. It also applies DS as neural network. DS-DQN does not support action masking.

In our previous study [15], additional RL algorithms have been evaluated, including those that did not use the DS neural network. However, our findings revealed that both DS algorithms consistently outperformed other RL methods. Thus, in this paper, we decided to narrow our evaluation scope and assess the performance of the two DS-based algorithms. The aim is to analyze their effectiveness and robustness under different reward strategies, facilitating a nuanced understanding of their applicability in real-world deployment scenarios. Through this extensive evaluation, we seek to provide valuable insights for researchers and practitioners seeking to leverage RL techniques for efficient deployment across K8s multi-clusters within the CC.

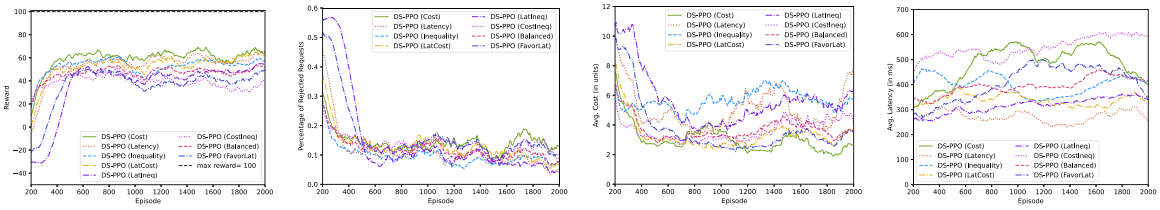
5.3. Cloud2Edge (C2E) application

Fig. 4 shows the deployment requirements for the various microservices of the C2E application applied in the evaluation of the *HephaestusForge* framework, based on the deployment specifications provided in the default K8s deployment files.[8] The C2E package provides a scalable, cloud-based Internet of Things (IoT) platform that connects sensor devices and processes their data using a Digital Twin (DT) platform. Its architecture includes several adapters alongside two main applications: Eclipse Hono and Eclipse Ditto. Eclipse Hono is an open-source framework that connects multiple IoT devices through remote service interfaces, facilitating communication between them using various protocols (e.g., HTTP REST, MQTT). It enables lower-end devices to connect to the Cloud backend to publish or report data such as telemetry. Additionally, Eclipse Hono supports updating the DT provided by Eclipse Ditto and other functional operations, such as sending commands and communicating events. Over the past few years, Eclipse Hono has been the focus of numerous studies (e.g., [48, 49]) due to its capabilities in device authentication and machine-to-machine management. Eclipse Ditto offers numerous services to implement DT for IoT devices, allowing the creation of IoT solutions without managing a custom backend. This enables users to concentrate on business requirements and application development. Ditto has been widely used in literature, often in combination with other systems and tools to create more sophisticated IoT environments [50,51]. We argue that C2E is an ideal application to test the proposed multi-cluster orchestration approach, as it includes multiple microservices designed to facilitate the efficient life-cycle management of IoT applications. Our framework is versatile and can support various applications without any modifications, unless adjustments to the observation space are required to incorporate additional parameters. In this evaluation, each request involves deploying several replicas of a single microservice related to a specific C2E functionality. The distribution of the requests follows randomly generated workload patterns, reenacting the dynamic fluctuations in demand that would be expected in CC scenarios. In addition, the RL agent does not utilize latency thresholds, but instead focuses on minimizing overall latency. Only certain baselines (described in Section 5.4) incorporate latency thresholds. The development of alternative reward strategies incorporating latency thresholds will be addressed in future work.

5.4. HephaestusForge framework

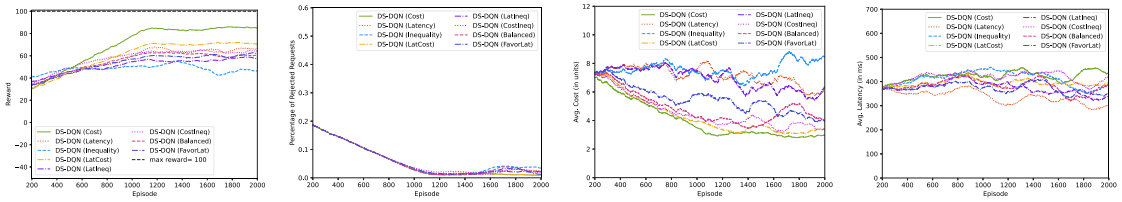
The *HephaestusForge* framework has been implemented in Python to ease the interaction with both the OpenAI Gym and the stable baselines 3 libraries. In the evaluation, an episode consists of 100 steps where the RL agent attempts to maximize the reward based on the current deployment request. To simulate dynamic latency conditions, when a replica is deployed on a cluster, the latency variables for that cluster are increased by a random factor of up to 15%. Conversely, when a replica is terminated after its mean service duration (defaulted to one time unit), the latency variables decrease by a random factor of up to 15%. This approach ensures that latency varies dynamically and uniquely in each episode, preventing any static patterns. This dynamic variation is crucial for the RL agent to learn that adding replicas to a cluster will result in increased latency. This approach reflects the typical trend of increasing latency as replicas are added to a cluster. The linear model assumes that each additional replica incrementally affects latency, a reasonable approximation for CC environments where network bandwidth or congestion increases with additional deployments. Similar relationships, such as linear, exponential, and random have been widely used in the literature [41,52–54] to simplify environments and facilitate the learning process of RL algorithms. During training, the multi-cluster infrastructure consists of four clusters. The RL algorithms have been executed on a 14-core Intel i7-12700H CPU @4.7 GHz processor with 16 GB of memory. The performance of the agents has been evaluated based on the following metrics:

[8] https://www.eclipse.org/packages/packages/cloud2edge/



(a) Accumulated Reward. (b) Rejected Requests. (c) Deployment Cost. (d) Avg. Latency.

Fig. 5. The training results for the DS-PPO agent evaluated for the multiple reward strategies.



(a) Accumulated Reward. (b) Rejected Requests. (c) Deployment Cost. (d) Avg. Latency.

Fig. 6. The training results for the DS-DQN agent evaluated for the multiple reward strategies.

Table 7
The execution time per episode (ep) during training.

Algorithm	Execution Time per ep (in s)	Execution Time for 2000 eps.
DS-PPO	0.799 ± 0.009	26.63 min
DS-DQN	0.494 ± 0.005	16.46 min
CPU-Greedy	0.116 ± 0.002	3.86 min
Binpack-Greedy	0.109 ± 0.002	3.63 min
Latency-Greedy	0.116 ± 0.002	3.86 min
Karmada-Greedy	**0.084** ± 0.001	**2.80** min

- **Accumulated reward** during each episode. It refers to the total sum of rewards obtained by an agent over time as it interacts with the environment.
- **Percentage of rejected requests** represented as $[0, 1]$. 1 corresponds to 100% rejection rate.
- **Average deployment cost** of each microservice request.
- **Average latency** expected for each accepted request.
- **Average CPU usage** (in %) of the selected cluster for the microservice deployment.
- **Gini Coefficient** highlighting the inequality of the deployment scheme, represented as $[0, 1]$.

Four heuristic-based baselines have also been evaluated to compare against the DS-based RL methods:

- **CPU-Greedy**: assigns all replicas to the cluster with the lowest resource consumption in terms of CPU usage.
- **Binpack-Greedy**: assigns all replicas to the cluster with the highest CPU allocation.
- **Latency-Greedy**: assigns all replicas to a cluster while adhering to the specified latency threshold.
- **Karmada-Greedy**: assigns all replicas to a cluster based on the cluster resource model (CPU and memory). It follows the default scheduling algorithm enabled in Karmada for dynamic replica assignment. Affinity and spread constraints (enabled via other scheduling plugins) are not considered.

6. Results

Time Complexity has been evaluated based on the training execution time for the DS-based RL agents for the *latency* strategy as shown in Table 7. The results highlight that training RL agents in near-real environments can speed up the training process, and consequently the applicability of RL methods in operational environments. Both algorithms require between 16 to 27 min to complete training over 2000 episodes. DS-DQN is slightly faster than DS-PPO, though both algorithms are significantly slower than all heuristic baselines. On average, heuristic algorithms complete 2000 episodes in about 2 to 4 min since no policy training is required.

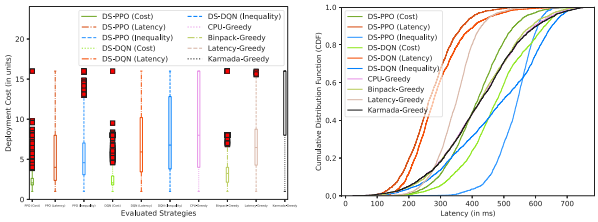
Training results for 2000 episodes are shown in Fig. 5 and Fig. 6 for all assessed reward strategies. A smoothing window of 200 episodes is applied to reduce spikes in the graphs. Despite fluctuations, all algorithms converge between the 700th and the 1000th episode, with some exhibiting slight improvements in rewards beyond this point. Most algorithms achieve high accumulated rewards for all reward strategies, though DS-DQN algorithms converge at a slightly higher state, resulting in higher rewards on average. It is noteworthy that all algorithms learn to minimize rejections, as reflected by their pursuit of higher rewards. PPO-based algorithms reject fewer than 20% of requests, while DQN-based algorithms reject fewer than 5%. Figs. 5(c), 5(d), 6(c), and 6(d) illustrate that the RL agents adhere to the favored strategy: cost-aware reward functions lead to lower deployment costs, and latency-aware reward functions result in lower latency. **Testing** has been executed for all strategies over 2000 episodes using the saved configuration from the training phase (after 2000 episodes). Table 8 summarizes the obtained results for the different algorithms based on the considered performance metrics. On average, latency-aware and cost-aware functions achieve higher rewards than inequality functions, highlighting the difficulty of achieving a fair distribution of microservice replicas. Most algorithms achieve a minimal rejection rate, with DS-PPO (*Inequality*) and DS-PPO (*LatIneq*) achieving as low as 0.002%. In contrast, heuristic baselines obtain higher rejection rates, ranging from 0.18% to 3.74%, because deploy-all actions might not be feasible at the end of episodes due to a lack of resources caused by previous actions. It is worth noting that DS-PPO achieves lower average rejection rates during testing compared to training. This difference occurs because, during training, the policy is stochastic: it samples actions based on the probabilities provided by the neural network. However, during testing, the policy is deterministic, selecting the action with the highest probability. This approach is standard in PPO implementations, as deterministically choosing the most probable action typically yields higher rewards, whereas stochasticity during training is necessary for effective exploration.

Table 8
Results obtained during the testing phase.

Alg.	Reward Function	Acc. Reward	Rejected Requests (in %)	Deployment Cost (in units)	Latency (in ms)	CPU Usage (in %)	Gini Coeff.
DS-PPO	*Latency*	73.1 ± 0.3	0.02 ± 0.01	5.87 ± 0.19	**268.6** ± 3.4	30.5 ± 0.5	0.67 ± 0.004
DS-DQN	*Latency*	70.1 ± 0.4	0.52 ± 0.05	7.03 ± 0.19	**290.7** ± 3.8	27.5 ± 0.5	0.55 ± 0.005
DS-PPO	*Cost*	**89.9** ± 0.4	0.05 ± 0.02	**2.50** ± 0.05	414.1 ± 4.1	38.5 ± 0.2	0.58 ± 0.005
DS-DQN	*Cost*	**89.7** ± 0.4	0.01 ± 0.01	**2.53** ± 0.05	477.1 ± 4.8	38.5 ± 0.2	0.55 ± 0.006
DS-PPO	*Inequality*	50.9 ± 0.5	**0.002** ± 0.004	5.57 ± 0.14	533.6 ± 2.7	31.5 ± 0.4	0.48 ± 0.005
DS-DQN	*Inequality*	49.4 ± 0.6	0.21 ± 0.03	8.01 ± 0.20	476.9 ± 6.4	25.9 ± 0.5	0.48 ± 0.006
DS-PPO	*LatCost*	75.2 ± 0.4	0.15 ± 0.03	3.59 ± 0.09	317.1 ± 3.4	35.9 ± 0.3	0.62 ± 0.005
DS-DQN	*LatCost*	74.3 ± 0.3	0.008 ± 0.007	2.57 ± 0.05	408.3 ± 5.1	38.5 ± 0.2	0.55 ± 0.006
DS-PPO	*LatIneq*	55.7 ± 0.3	**0.002** ± 0.004	5.51 ± 0.13	405.8 ± 2.0	31.6 ± 0.4	0.47 ± 0.005
DS-DQN	*LatIneq*	58.5 ± 0.3	1.42 ± 0.07	6.67 ± 0.16	307.9 ± 3.8	27.3 ± 0.4	0.46 ± 0.005
DS-PPO	*CostIneq*	55.3 ± 0.6	0.003 ± 0.004	4.51 ± 0.14	573.7 ± 3.9	34.0 ± 0.4	0.66 ± 0.004
DS-DQN	*CostIneq*	64.4 ± 0.4	0.003 ± 0.004	2.80 ± 0.07	511.2 ± 4.4	38.0 ± 0.3	0.58 ± 0.006
DS-PPO	*Balanced*	59.4 ± 0.2	0.003 ± 0.004	3.44 ± 0.08	501.5 ± 2.8	36.7 ± 0.3	0.51 ± 0.006
DS-DQN	*Balanced*	63.2 ± 0.3	0.26 ± 0.05	3.07 ± 0.08	392.0 ± 3.5	37.0 ± 0.3	0.54 ± 0.006
DS-PPO	*FavorLat*	65.1 ± 0.3	0.008 ± 0.008	3.28 ± 0.08	334.3 ± 3.1	36.8 ± 0.3	0.58 ± 0.005
DS-DQN	*FavorLat*	63.3 ± 0.3	0.003 ± 0.004	3.59 ± 0.09	378.9 ± 3.4	36.1 ± 0.3	0.52 ± 0.006
CPU	–	–	0.18 ± 0.07	9.63 ± 0.25	422.8 ± 5.8	**18.5** ± 0.4	0.72 ± 0.003
Binpack	–	–	0.18 ± 0.07	3.67 ± 0.10	420.7 ± 5.6	34.6 ± 0.3	0.66 ± 0.004
Latency	–	–	3.74 ± 0.45	6.71 ± 0.14	347.8 ± 2.9	25.5 ± 0.3	**0.26** ± 0.005
Karmada	–	–	0.18 ± 0.07	12.25 ± 0.21	424.1 ± 5.9	**14.8** ± 0.4	0.73 ± 0.002

As expected, cost-aware reward functions result in lower average deployment costs of 2.5 units as shown in Fig. 7(a), while latency-aware reward functions lead to lower latency, ranging from 268.6 to 290.7 ms as demonstrated in Fig. 7(b). Regarding CPU usage, the CPU-Greedy and Karmada-Greedy approaches achieve the minimum CPU usage for the selected cluster. The Latency-Greedy approach achieves the lowest Gini Coefficient of 0.26, with most inequality-aware algorithms following closely with coefficients between 0.46 and 0.58. These results highlight the differences in service placement based on the chosen objective. The DS-DQN (*Balanced*) achieves a good compromise among cost (3.07 units), latency (392.0 ms), and inequality (0.54). These outcomes confirm that RL-based approaches offer significant advantages over heuristic methods, albeit at the cost of increased training time. While substantial progress has been made, improving the efficiency of the training process remains a key focus for future work. We argue that the characteristics of the DS methodology provide notable benefits in terms of time efficiency for CC orchestration. In particular, its ability to handle inputs and outputs of arbitrary sizes is crucial, enabling seamless application to scenarios with varying numbers of managed clusters.

Multiple Reward Strategies have been evaluated to assess their impact on performance. Choosing an appropriate reward strategy is crucial, as it guides the learning process and significantly affects the placement strategy. As the complexity of the problem increases, it is well known that approaches such as metaheuristics can struggle, particularly when confronted with multiple performance factors, as is the case in this work. In contrast, our evaluation shows that RL identifies efficient policies that balance various performance factors simultaneously. However, further analysis is needed to assess how close these RL policies are to optimal solutions. The comparison of the results obtained from *HephaestusForge* with those derived from exact or approximation algorithms is planned for future work, establishing benchmarks and evaluating our strategies against known optimal or near-optimal solutions. **Scalability** has been assessed by increasing the number of microservice replicas per request, resulting in higher ratios during the experiments. The number of available clusters has been maintained at four, while the number of microservice replicas varies from 4 to 32, creating the following ratios $[1, 2, 3, 4, 6, 8]$. The minimum and maximum number of replicas has been fixed for this experiment to achieve the desired ratio. Fig. 8 shows that all DS-based algorithms accepted most requests, even at high ratios (greater than 6). In contrast, most heuristic baselines already exhibited a 40% rejection rate at ratios of 3 and 4. In addition, as the ratio increased, most RL agents struggled



(a) Deployment Cost (in units). (b) Avg. Latency (in ms).

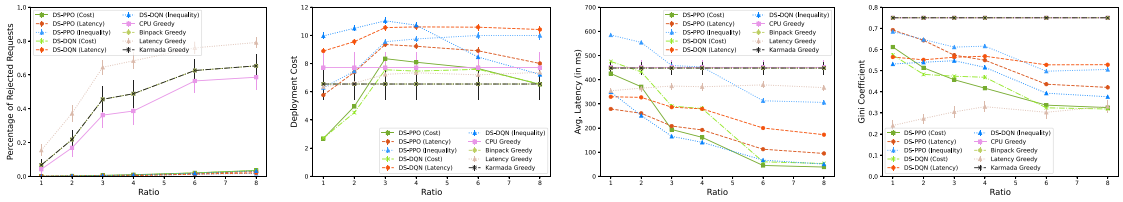
Fig. 7. The expected latency and deployment cost during testing.

to minimize deployment costs while accepting the majority of requests, although latency slightly decreased due to the higher rejection rate. The Gini coefficient also decreases with the ratio, indicating a fairer distribution as more microservice replicas need deployment.

Increasing the number of clusters has been evaluated by varying the cluster size from 4 to 32 and setting the minimum and maximum replicas equal to the number of clusters (c) and four times the number of clusters ($4 \times c$), respectively. This maintains a ratio between 1 and 4 throughout the experiment. The results demonstrate the enormous potential of the DS neural network. All RL agents can find near-optimal allocation schemes for all strategies, even when trained in a small-scale setup as shown in Fig. 9. As the number of clusters increases, all algorithms converge to a similar state, reflecting the challenge of optimizing cost or latency under these conditions. Nonetheless, cost-aware algorithms and latency-aware agents achieve the lowest deployment costs and lower latency, respectively. DS-based algorithms can effectively optimize microservice placement in a multi-cluster setup higher than the trained scenario, without needing retraining.

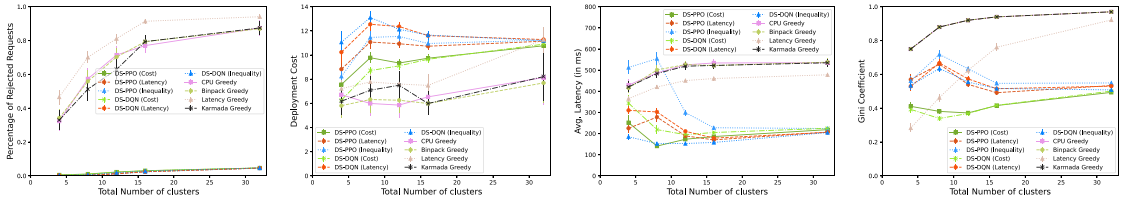
What if additional spreading strategies are included in the RL action space? As a final step in the evaluation, this study examines the implications of expanding the action space of the RL agents with two extra spreading actions:

- **First Fit Increasing (FFI)** acts similarly to the previously presented FFD heuristic but sorts instances in increasing order before placement.
- **Best Fit (BF1B1)** allocates each replica to the cluster that minimizes remaining space in terms of computing resources after placement, sorting replicas individually.



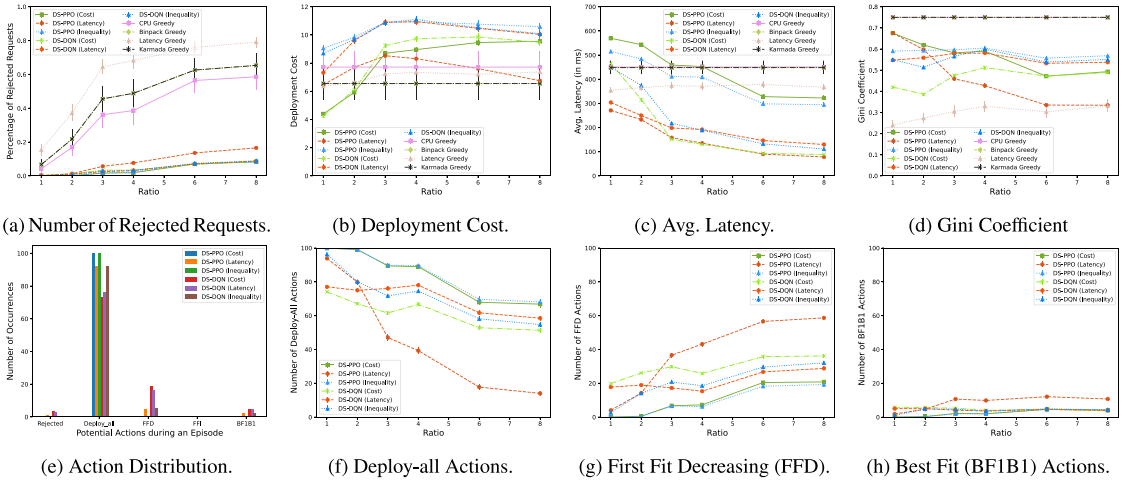
(a) Number of Rejected Requests. (b) Deployment Cost. (c) Avg. Latency. (d) Gini Coefficient

Fig. 8. The results for the trained DS-based algorithms while varying the number of deployed replicas per request.



(a) Number of Rejected Requests. (b) Deployment Cost. (c) Avg. Latency. (d) Gini Coefficient

Fig. 9. The results for the trained DS-based algorithms while varying the number of available clusters.



(a) Number of Rejected Requests. (b) Deployment Cost. (c) Avg. Latency. (d) Gini Coefficient

(e) Action Distribution. (f) Deploy-all Actions. (g) First Fit Decreasing (FFD). (h) Best Fit (BF1B1) Actions.

Fig. 10. The results for the trained DS-based algorithms while varying the number of deployed replicas per request for the extended version of the action space.

These two additional spreading strategies have been implemented and added to the action space of *HephaestusForge*, further extending the action possibilities of the RL agents. A scalability test, similar to the previously conducted one, has been carried out with an increase in the number of microservice replicas per request, resulting in higher ratios during the experiment. Fig. 10 illustrates the performance of the RL agents and heuristic baselines during testing (training for 2000 episodes has been conducted but results are omitted to avoid duplication). Fig. 10(e) shows the distribution of action selections throughout the experiment. Deploy-all actions are more commonly selected, while spreading actions are less frequent. However, based on Figs. 10(f), 10(g), and 10(h), spreading actions are increasingly selected as the ratio grows during the experiment. Computing resources become scarcer towards the end of episodes when the ratio is high, making the agent adapt its policy and select more spreading actions such as FFD and BF1B1. Interestingly, the agent never selects FFI, indicating a calculated avoidance of this action during testing. In terms of performance, the RL agents achieve slightly lower latency on average, slightly higher deployment costs, and a higher rejection rate compared to the RL agents tested with a smaller action space (Fig. 8).

In summary, this paper investigates efficient multi-cluster orchestration strategies, focusing on the well-known K8s platform and recent trends in RL. A multi-objective reward function demonstrates that RL algorithms can find suitable actions to maximize accumulated rewards based on chosen goals. The *HephaestusForge* framework validated the RL approach, showcasing that training RL in near-real environments accelerates the training process. Our RL-based approach benefits the Karmada multi-cluster orchestration solution by effectively balancing the trade-off between deploying all replicas in a single cluster and distributing them across multiple clusters for various placement objectives. During the testing phase, all DS-based algorithms achieved high rewards for the evaluated strategies compared to existing heuristic baselines, highlighting the potential of RL for multi-cluster orchestration. Additionally, the DS neural network has demonstrated its enormous potential, as it can be directly applied to different multi-cluster environments with varying cluster sizes. Without DS, RL algorithms require retraining for each specific cluster size, which is considerably more costly.

6.1. Discussion on performance differences between DS-PPO and DS-DQN

Exploration vs. Exploitation As a policy gradient method, DS-PPO continuously updates the policy to maximize the expected reward directly. It naturally balances exploration and exploitation by adjusting the policy based on observed rewards, allowing it to explore a wider range of actions while refining its policy more effectively. This is particularly beneficial in environments with large action space or complex dynamics, such as the multi-cluster orchestration problem where optimal actions vary based on multiple factors (e.g., latency, cost, inequality). DS-DQN, being a value-based method, estimates the value of taking a certain action in a particular state and selects actions that maximize this value. While effective, DS-DQN might struggle with exploration in high-dimensional action spaces, potentially leading to suboptimal performance if the state–action space is not fully explored. This can be a limiting factor when dealing with complex objectives such as multi-objective optimization in multi-cluster environments.

Scalability and Convergence DS-PPO's policy optimization approach is known for its stable learning process, which can lead to faster convergence in environments with a large number of potential actions and states. This stability is crucial in scenarios where the agent must adapt to varying conditions, such as different cluster sizes or varying ratios of microservice replicas, as tested in our scalability experiments. DS-DQN might take longer to converge or might converge to suboptimal policies, especially in environments with high variability or where the policy needs to generalize across a wide range of conditions. This could explain why DS-DQN might underperform in scenarios where rapid adaptation is required.

Handling Continuous and Discrete Action Spaces DS-PPO is inherently well suited for environments that may involve continuous action spaces or where fine-tuning actions lead to better results. Although our environment primarily deals with discrete actions, the flexibility of DS-PPO in handling different types of actions might give it an edge in situations where more granular control over decisions is beneficial. DS-DQN is primarily designed for discrete action spaces, and while it can be extended to handle continuous actions, the original formulation might be less flexible in environments where fine distinctions between actions are crucial for performance.

Adaptation to New Strategies In the extended action space scenario, where additional spreading strategies such as FFI and BF1B1 are introduced, DS-PPO might be better to adapt to these new strategies due to its constant updates of policy and inherent flexibility. This adaptation is reflected in the more nuanced action selection observed in the experiments. DS-DQN might require more episodes to effectively incorporate new strategies into its policy, leading to less optimal performance when the action space is expanded. This could manifest as higher deployment costs or a higher rejection rate, as observed in our results.

7. Open challenges & future directions

This section discusses the remaining hurdles in orchestration in modern cloud platforms, also highlighting future directions. **Inter-Cluster Interoperability** is a crucial aspect of multi-cluster orchestration that ensures diverse clusters can work together seamlessly, regardless of their underlying infrastructure or platform. Effective interoperability allows companies to leverage the strengths of different cloud providers, data centers, and K8s distributions while maintaining a unified operational framework. However, different K8s distributions (e.g., K3s,[9] and MicroK8s[10]) may have unique features, configurations, and extensions that complicate interoperability. Achieving compatibility requires adherence to core K8s APIs and avoiding proprietary extensions. Existing tools such as Open Policy Agent (OPA)[11] and Kyverno[12] help enforce uniform policies across clusters, but further research is needed for fully autonomous multi-cluster management, as current policy selection remains largely manual by cloud administrators.

Security in a multi-cluster environment is a complex yet critical aspect of inter-cluster interoperability. Each cluster may have different security postures, requiring consistent management of authentication, authorization, secrets, and network security policies. Applications across clusters must adhere to compliance regulations to protect sensitive data. Implementing network segmentation and policies, facilitated by network policy tools such as Cilium,[13] helps control traffic flow and restrict communication, minimizing the risk of unauthorized access and breaches. **Efficient Configuration Management** is essential for maintaining consistency, reliability, and operational efficiency in multi-cluster environments. As organizations scale their infrastructure across multiple clusters, managing configurations becomes increasingly complex. Effective configuration management involves using standardized tools and practices to handle the diverse configurations required for different clusters, applications, and environments. Declarative configuration simplifies this process by allowing easy version control and replication. For example, K8s YAML manifests enable the declaration of resource specifications that K8s uses to manage the state of the cluster. Also, Helm[14] enables the creation of reusable templates for consistent and customizable configurations, reducing errors and enhancing reliability in multi-cluster deployments.

Refined multi-objective formalizations can significantly enhance solution quality by providing a more comprehensive perspective on the trade-offs among conflicting objectives, typical within CC scenarios. Traditional approaches, such as scalarization or lexicographic ordering, often prove infeasible due to the difficulty of specifying quantitative preferences between heterogeneous objectives a priori [55]. Due to the high computational complexity for solving multi-objective formulations optimally, heuristics and genetic algorithms are frequently used to obtain approximate solutions in reasonable computational times [56]. Multi-objective RL also offers a promising alternative by enabling the exploration of the Pareto Front to identify balanced solutions [57]. We will study Pareto Q-learning in future work, an RL-based approach that extends traditional Q-learning by using Q-values as vectors representing multiple objectives, allowing the agent to approximate the Pareto Front effectively. Additionally, policy gradient methods adapted for Pareto optimization will be explored, leveraging their potential to balance exploration and exploitation of the Pareto Front. **Real-world evaluation** is essential for validating RL-based strategies in K8s environments. As part of future work, we plan to assess the performance of *HephaestusForge* in an operational K8s setting. This will involve exploring workload patterns derived from actual C2E application usage data or other cloud–edge application profiles to evaluate the generalization capabilities of *HephaestusForge*. Moreover, this evaluation will provide an opportunity to establish a more accurate, empirically driven relationship between the number of deployed replicas and their end-to-end latency, facilitating the development of a more robust and realistic latency model for integration into our framework.

In summary, this section emphasizes the need for extensive research in network management for multi-cluster orchestration to tackle interoperability and security challenges in distributed cloud environments. Ongoing research, interdisciplinary collaborations, and standardized best practices are essential elements in the journey towards more efficient and reliable multi-cluster infrastructures. All these efforts will contribute to overcoming the complexity of multi-cluster orchestration and improve the performance and security of modern applications.

[9] https://k3s.io/
[10] https://microk8s.io/
[11] https://www.openpolicyagent.org/
[12] https://kyverno.io/
[13] https://cilium.io/
[14] https://helm.sh/

8. Conclusions

This paper studies the efficient scheduling of microservices in a multi-cluster scenario within the CC. An RL-based approach inspired on the OpenAI Gym library has been proposed to handle efficient multi-cluster orchestration in the well-known K8s platform. The evaluation considers a multi-objective reward function that demonstrates the feasibility of RL for the multi-cluster orchestration problem addressed in the paper. Results show that scalable and generalizable policies can be attainable by incorporating the DS neural network in typical RL algorithms, achieving high performance for scenarios higher than the trained one compared to heuristic baselines, without the need for retraining these algorithms. DS-based algorithms achieve minimal rejection rates (as low as 0.002%, 90x less than the baseline Karmada scheduler). Cost-aware strategies result in lower deployment costs (2.5 units), and latency-aware functions achieve lower latency (268–290 ms), improving by 1.5x and 1.3x, respectively, over the best-performing baselines. Multi-agent and distributed RL scenarios will be studied as future work to find optimal combinations of opposing scheduling strategies. This work contributes to the field by providing the *HephaestusForge* framework released in open-source, allowing researchers to evaluate placement policies, and potentially guide the development of additional scheduling algorithms.

CRediT authorship contribution statement

José Santos: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Mattia Zaccarini:** Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Filippo Poltronieri:** Writing – review & editing, Writing – original draft, Methodology, Investigation, Conceptualization. **Mauro Tortonesi:** Writing – review & editing, Supervision, Project administration, Funding acquisition, Conceptualization. **Cesare Stefanelli:** Writing – review & editing, Supervision. **Nicola Di Cicco:** Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Filip De Turck:** Writing – review & editing, Supervision, Project administration.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Jose Santos reports financial support was provided by Research Foundation Flanders (FWO), grant number 1299323N. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially supported by the Spoke 1 “FutureHPC & BigData” of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 - Next Generation EU (NGEU). José Santos is funded by the Research Foundation Flanders (FWO), Belgium, grant number 1299323N.

Data availability

The framework will be released in open-source. Details in the paper.

References

[1] C. Pahl, A. Brogi, J. Soldani, P. Jamshidi, Cloud container technologies: a state-of-the-art review, IEEE Trans. Cloud Comput. 7 (3) (2017) 677–692.
[2] Y. Gan, C. Delimitrou, The architectural implications of cloud microservices, IEEE Comput. Archit. Lett. 17 (2) (2018) 155–158.
[3] J. Santos, T. Wauters, B. Volckaert, F. De Turck, Fog computing: Enabling the management and orchestration of smart city applications in 5G networks, Entropy 20 (1) (2017) 4.
[4] D. Zhao, G. Sun, D. Liao, S. Xu, V. Chang, Mobile-aware service function chain migration in cloud–fog computing, Future Gener. Comput. Syst. 96 (2019) 591–604.
[5] K. Cao, Y. Liu, G. Meng, Q. Sun, An overview on edge computing research, IEEE Access 8 (2020) 85714–85728.
[6] G. Sun, Y. Li, Y. Li, D. Liao, V. Chang, Low-latency orchestration for workflow-oriented service function chain in edge computing, Future Gener. Comput. Syst. 85 (2018) 116–128.
[7] D. Balouek-Thomert, E.G. Renart, A.R. Zamani, A. Simonet, M. Parashar, Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows, Int. J. High Perform. Comput. Appl. 33 (6) (2019) 1159–1174.
[8] P. Bellavista, N. Bicocchi, M. Fogli, C. Giannelli, M. Mamei, M. Picone, Exploiting microservices and serverless for digital twins in the cloud-to-edge continuum, Future Gener. Comput. Syst. 157 (2024) 275–287.
[9] C. Badue, R. Guidolini, R.V. Carneiro, P. Azevedo, V.B. Cardoso, A. Forechi, L. Jesus, R. Berriel, T.M. Paixao, F. Mutz, et al., Self-driving cars: A survey, Expert Syst. Appl. 165 (2021) 113816.
[10] D.C. Nguyen, M. Ding, P.N. Pathirana, A. Seneviratne, J. Li, D. Niyato, O. Dobre, H.V. Poor, 6G Internet of Things: A comprehensive survey, IEEE Internet Things J. (2021).
[11] J. Santos, T. Wauters, B. Volckaert, F. De Turck, Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions, IEEE Commun. Surv. Tutor. 23 (4) (2021) 2557–2589.
[12] Z. Zhong, R. Buyya, A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources, ACM Trans. Internet Technol. (TOIT) 20 (2) (2020) 1–24.
[13] Y. Han, S. Shen, X. Wang, S. Wang, V.C. Leung, Tailored learning-based scheduling for kubernetes-oriented edge-cloud system, in: IEEE INFOCOM 2021-IEEE Conference on Computer Communications, IEEE, 2021, pp. 1–10.
[14] B. Burns, J. Beda, K. Hightower, Kubernetes: Up and Running: Dive into the Future of Infrastructure, O’Reilly Media, 2019.
[15] J. Santos, M. Zaccarini, F. Poltronieri, M. Tortonesi, C. Stefanelli, N. Di Cicco, F. De Turck, Efficient microservice deployment in kubernetes multi-clusters through reinforcement learning, in: NOMS2024, the IEEE/IFIP Network Operations and Management Symposium, IEEE, 2024, pp. 1–9.
[16] Karmada documentation, 2023, [Online]. Available: https://karmada.io/. (Accessed on 26 June 2023).
[17] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R.R. Salakhutdinov, A.J. Smola, Deep sets, in: Advances in Neural Information Processing Systems, vol. 30, 2017.
[18] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, H.A. Chan, Optimal virtual network function placement in multi-cloud service function chaining architecture, Comput. Commun. 102 (2017) 1–16.
[19] C. Guerrero, I. Lera, C. Juiz, Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications, J. Supercomput. 74 (7) (2018) 2956–2983.
[20] S.K. Panda, I. Gupta, P.K. Jana, Task scheduling algorithms for multi-cloud systems: allocation-aware approach, Inform. Syst. Front. 21 (2019) 241–259.
[21] S. Lee, S. Son, J. Han, J. Kim, Refining micro services placement over multiple kubernetes-orchestrated clusters employing resource monitoring, in: 2020 IEEE 40th International Conference on Distributed Computing Systems, ICDCS, IEEE, 2020, pp. 1328–1332.
[22] F. Rossi, V. Cardellini, F.L. Presti, M. Nardelli, Geo-distributed efficient deployment of containers with kubernetes, Comput. Commun. 159 (2020) 161–174.
[23] Y. Zhang, B. Di, Z. Zheng, J. Lin, L. Song, Distributed multi-cloud multi-access edge computing by multi-agent reinforcement learning, IEEE Trans. Wireless Commun. 20 (4) (2020) 2565–2578.
[24] M.A. Tamiru, G. Pierre, J. Tordsson, E. Elmroth, mck8s: An orchestration platform for geo-distributed multi-cluster environments, in: 2021 International Conference on Computer Communications and Networks, ICCCN, IEEE, 2021, pp. 1–10.
[25] T. Shi, H. Ma, G. Chen, S. Hartmann, Location-aware and budget-constrained service brokering in multi-cloud via deep reinforcement learning, in: H. Hacid, O. Kao, M. Mecella, N. Moha, H.-y. Paik (Eds.), Service-Oriented Computing, Springer International Publishing, Cham, 2021, pp. 756–764.

[26] S. Qin, D. Pi, Z. Shao, Y. Xu, Y. Chen, Reliability-aware multi-objective memetic algorithm for workflow scheduling problem in multi-cloud system, IEEE Trans. Parallel Distrib. Syst. 34 (4) (2023) 1343–1361, http://dx.doi.org/10.1109/TPDS.2023.3245089.

[27] R. Moreno-Vozmediano, R.S. Montero, E. Huedo, I.M. Llorente, Intelligent resource orchestration for 5G edge infrastructures, Future Internet 16 (3) (2024) 103.

[28] A. Suzuki, M. Kobayashi, E. Oki, Multi-agent deep reinforcement learning for cooperative computing offloading and route optimization in multi cloud-edge networks, IEEE Trans. Netw. Serv. Manag. (2023).

[29] M. Zaccarini, B. Cantelli, M. Fazio, W. Fornaciari, F. Poltronieri, C. Stefanelli, M. Tortonesi, et al., VOICE: Value-of-information for compute continuum ecosystems, in: 2024 27th Conference on Innovation in Clouds, Internet and Networks, ICIN ICIN 2024, IEEE, 2024, pp. 142–149.

[30] S. Ejaz, A.-N. Mays, FORK: A kubernetes-compatible federated orchestrator of fog-native applications over multi-domain edge-to-cloud ecosystems, in: 2024 27th Conference on Innovation in Clouds, Internet and Networks, ICIN ICIN 2024, IEEE, 2024.

[31] B. Varghese, R. Buyya, Next generation cloud computing: New trends and research directions, Future Gener. Comput. Syst. 79 (2018) 849–861.

[32] M.S. Raghavendra, P. Chawla, A. Rana, A survey of optimization algorithms for fog computing service placement, in: 2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions), ICRITO, IEEE, 2020, pp. 259–262.

[33] M. Taghavian, Y. Hadjadj-Aoul, G. Texier, N. Huin, P. Bertin, An approach to network service placement reconciling optimality and scalability, IEEE Trans. Netw. Serv. Manag. 20 (3) (2023) 2218–2229.

[34] J. Santos, J. van der Hooft, M.T. Vega, T. Wauters, B. Volckaert, F. De Turck, Efficient orchestration of service chains in fog computing for immersive media, in: 2021 17th International Conference on Network and Service Management, CNSM, IEEE, 2021, pp. 139–145.

[35] J. Santos, T. Wauters, B. Volckaert, F. De Turck, gym-hpa: Efficient auto-scaling via reinforcement learning for complex microservice-based applications in kubernetes, in: NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium, IEEE, 2023, pp. 1–9.

[36] J. Santos, E. Reppas, T. Wauters, B. Volckaert, F. De Turck, Gwydion: Efficient auto-scaling for complex containerized applications in kubernetes through reinforcement learning, J. Netw. Comput. Appl. (2024) 104067, http://dx.doi.org/10.1016/j.jnca.2024.104067.

[37] M. Fogli, T. Kudla, B. Musters, G. Pingen, C. Van den Broek, H. Bastiaansen, N. Suri, S. Webb, Performance evaluation of kubernetes distributions (k8s, k3s, kubeedge) in an adaptive and federated cloud infrastructure for disadvantaged tactical networks, in: 2021 International Conference on Military Communication and Information Systems, ICMCIS, IEEE, 2021, pp. 1–7.

[38] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, 2016, arXiv preprint arXiv:1606.01540.

[39] J. Santos, C. Wang, T. Wauters, F. De Turck, Diktyo: Network-aware scheduling in container-based clouds, IEEE Trans. Netw. Serv. Manag. (2023).

[40] O. Mart, C. Negru, F. Pop, A. Castiglione, Observability in kubernetes cluster: Automatic anomalies detection using prometheus, in: 2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems, HPCC/SmartCity/DSS, IEEE, 2020, pp. 565–570.

[41] N.D. Cicco, G.F. Pittalà, G. Davoli, D. Borsatti, W. Cerroni, C. Raffaelli, M. Tornatore, DRL-FORCH: A scalable deep reinforcement learning-based fog computing orchestrator, in: 2023 IEEE 9th International Conference on Network Softwarization, NetSoft, 2023, pp. 125–133, http://dx.doi.org/10.1109/NetSoft57336.2023.10175398.

[42] Amazon AWS, Amazon EC2 on-demand pricing, 2023, [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/. (Accessed on 28 September 2023).

[43] H. Sami, A. Mourad, H. Otrok, J. Bentahar, Demand-driven deep reinforcement learning for scalable fog and service placement, IEEE Trans. Serv. Comput. 15 (5) (2021) 2671–2684.

[44] S. Huang, S. Ontañón, A closer look at invalid action masking in policy gradient algorithms, 2020, arXiv preprint arXiv:2006.14171.

[45] K. Blesch, O.P. Hauser, J.M. Jachimowicz, Measuring inequality beyond the gini coefficient may clarify conflicting findings, Nat. Hum. Behav. 6 (11) (2022) 1525–1536.

[46] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, N. Dormann, Stable-baselines3: Reliable reinforcement learning implementations, J. Mach. Learn. Res. 22 (268) (2021) 1–8.

[47] S. Huang, R.F.J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, J.G. Araújo, CleanRL: High-quality single-file implementations of deep reinforcement learning algorithms, J. Mach. Learn. Res. 23 (274) (2022) 1–18, URL http://jmlr.org/papers/v23/21-1342.html.

[48] M. Aly, F. Khomh, S. Yacout, Kubernetes or OpenShift? Which technology best suits eclipse hono IoT deployments, in: 2018 IEEE 11th Conference on Service-Oriented Computing and Applications, SOCA, 2018, pp. 113–120, http://dx.doi.org/10.1109/SOCA.2018.00024.

[49] J. Lee, S. Kang, I.-G. Chun, MIoTwins: Design and evaluation of MIoT framework for private edge networks, in: 2021 International Conference on Information and Communication Technology Convergence, ICTC, 2021, pp. 1882–1884, http://dx.doi.org/10.1109/ICTC52510.2021.9621144.

[50] Y. Lim, Y.K. Lee, J. Yoo, D. Yoon, An open source-based digital twin broker interface for interaction between real and virtual assets, in: 2022 13th International Conference on Information and Communication Technology Convergence, ICTC, 2022, pp. 1657–1659, http://dx.doi.org/10.1109/ICTC55196.2022.9952499.

[51] J. Kristan, P. Azzoni, L. Römer, S.E. Jeroschewski, E. Londero, Evolving the ecosystem: Eclipse arrowhead integrates eclipse IoT, in: NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium, 2022, pp. 1–6, http://dx.doi.org/10.1109/NOMS54207.2022.9789922.

[52] D. Lee, J.-H. Yoo, J.W.-K. Hong, Deep q-networks based auto-scaling for service function chaining, in: 2020 16th International Conference on Network and Service Management, CNSM, IEEE, 2020, pp. 1–9.

[53] A. F. Ocampo, J. Santos, Reinforcement learning-driven service placement in 6G networks across the compute continuum, in: 2024 20th International Conference on Network and Service Management, CNSM, IEEE, 2024, pp. 1–9.

[54] J. Santos, T. Wauters, F. De Turck, P. Steenkiste, Towards optimal load balancing in multi-zone kubernetes clusters via reinforcement learning, in: 2024 33rd International Conference on Computer Communications and Networks, ICCCN, IEEE, 2024, pp. 1–9.

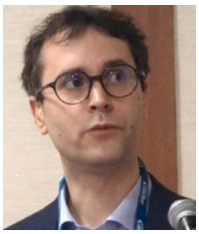
[55] N. Di Cicco, F. Poltronieri, J. Santos, M. Zaccarini, M. Tortonesi, C. Stefanelli, F. De Turck, Multi-objective scheduling and resource allocation of kubernetes replicas across the compute continuum, in: 2024 20th International Conference on Network and Service Management, CNSM, IEEE, 2024, pp. 1–9.

[56] Y. Guo, B. Liu, W. Lin, X. Ye, J.Z. Wang, Dynamic neighborhood grouping-based multi-objective scheduling algorithm for workflow in hybrid cloud, Future Gener. Comput. Syst. (2024) 107633, http://dx.doi.org/10.1016/j.future.2024.107633.

[57] C.F. Hayes, R. Rădulescu, E. Bargiacchi, J. Källström, M. Macfarlane, M. Reymond, T. Verstraeten, L.M. Zintgraf, R. Dazeley, F. Heintz, E. Howley, A.A. Irissappane, P. Mannion, A. Nowé, G. Ramos, M. Restelli, P. Vamplew, D.M. Roijers, A practical guide to multi-objective reinforcement learning and planning, Auton. Agents Multi-Agent Syst. 36 (1) (2022) http://dx.doi.org/10.1007/s10458-022-09552-y.

José Santos obtained his M.Sc. degree in Electrical and Computers Engineering in July 2015 from the University of Porto, Portugal. Recently, he completed his doctoral studies at Ghent University in April 2022. He is currently a Postdoctoral Researcher in the Internet Technology and Data Science Lab (IDLab) Research Group at Ghent University - imec, Belgium. His research interests include Cloud and Fog Computing, IoT, Service Function Chaining, and Reinforcement Learning. His work has been published in more than 25 scientific publications. He received the Ph.D. Excellence Award from imec in 2022 and the Best Dissertation Award at NOMS 2023 based on the research conducted during his Ph.D. about efficient orchestration strategies in Fog Computing.

Mattia Zaccarini is a Ph.D. student at the Engineering Department of the University of Ferrara. He obtained his Bachelor's degree in Electronics and Computer Science Engineering in 2018 and his Master's degree in Computer and Automation Engineering in 2022 from the University of Ferrara. He is currently part of the Big Data and Compute Continuum research laboratory and his research activity is focused on Compute Continuum, Network Digital Twin, Reinforcement Learning and Computational Intelligence techniques.



Filippo Poltronieri received the Ph.D. degree from the University of Ferrara, Italy, in 2021. He is currently an Assistant Professor with the Department of Engineering, University of Ferrara. His research interests include distributed systems, optimization techniques for network and service management, compute continuum, and tactical networks. He has been visiting the Florida Institute for Human & Machine Cognition (IHMC) in Pensacola, FL (USA) in 2016–2017 and 2018.

Mauro Tortonesi (Member, IEEE) is an Associate Professor and the head of the Big Data and Compute Continuum research laboratory at the University of Ferrara, Italy. He received the Ph.D. degree in computer engineering from the University of Ferrara, in 2006. He was a Visiting Scientist with the Florida Institute for Human & Machine Cognition (IHMC), Pensacola, FL, USA, from 2004 to 2005 and with the United States Army Research Laboratory, Adelphi, MD, USA, in 2015. He participates/has participated with several roles in a wide number of research projects in the distributed systems area, with particular reference to Compute Continuum, Big Data, and IoT solutions in industrial and military environments. He has coauthored over 100 publications and has 4 international patents.

Cesare Stefanelli received the Ph.D. degree in computer science engineering from the University of Bologna, Italy, in 1996. He is currently a Full Professor of distributed systems with the Engineering Department, University of Ferrara, Italy. At the University of Ferrara he coordinates a Technopole Laboratory dealing with industrial research and technology transfer. He holds several patents, and coordinates industrial research projects carried on in collaboration with several companies. His research interests include distributed and mobile computing in wireless and ad hoc networks, network and systems management, and network security.

Nicola Di Cicco is currently working toward the Ph.D. degree with the Department of Electronic, Information, and Bioengineering, Politecnico di Milano, Milan, Italy. His research interests include deep learning and reinforcement learning for network optimization and control.

Filip De Turck leads the network and service management research group at Ghent University, Belgium and imec. He (co-) authored over 700 peer reviewed papers and his research interests include design of efficient softwarized network and cloud systems. He is involved in several research projects with industry and academia, served as chair of the IEEE Technical Committee on Network Operations and Management (CNOM), and serves as a steering committee member of the IM, NOMS, CNSM and NetSoft conferences. Prof. Filip De Turck served as Editor-in-Chief of IEEE Transactions on Network and Service Management (TNSM), and was named an IEEE Fellow since 2021.