GreenWise: Intelligent Application Migration for Containerized Machine Learning Services in the Computing Continuum

Anonymous Author(s)

Abstract

Machine Learning (ML) services require both responsiveness and efficiency. In the dynamic Computing Continuum (CC), service migration has emerged as an important strategy to optimize service performance, improve power efficiency, or reduce operational costs. A basis enabler of the service migration in the CC is containerization and container orchestration, however, dynamically moving the service among Cloud and Edge servers based on several uncertainties is still challenging. In this paper, we present GreenWise, a framework for intelligent service migration for containerized ML services in the heterogeneous CC. GreenWise extends the monitoring agents to continuously monitor power consumption and performance metrics across diverse layers and sources in real-time, providing a holistic view of states. Leveraging Reinforcement Learning (RL), it enables Power-Performance aware strategies to achieve near-optimal online migration decisions under dynamic conditions. We perform GreenWise on top of a Kubernetes-based CC platform, implementing agents for intelligent migration for containerized ML services. Experimental results demonstrate that GreenWise achieves effective trade-offs between performance and power efficiency. Our proposed *Power-Latency-Aware (MaskPPO)* migration strategy outperforms *Random* and *Round-Robin* baselines 159.2% and 13.8% in a real cluster. These results highlight GreenWise's potential for sustainable and intelligent ML service migration in the dynamic CC.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Computing methodologies → Planning and scheduling**; **Machine learning algorithms**; • **Software and its engineering → Power management**.

Keywords

Intelligent Service Migration, Computing Continuum, Kubernetes, Power-Performance Aware, Reinforcement Learning.

ACM Reference Format:
Anonymous Author(s). 2025. GreenWise: Intelligent Application Migration for Containerized Machine Learning Services in the Computing Continuum. In *Proceedings of IEEE/ACM International Conference on Utility and Cloud Computing (UCC2025)*. ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Modern computing infrastructure is evolving at a fast pace to a distributed Computing Continuum (CC), which seamlessly integrates the computation power of the Cloud servers and low-latency and context power-aware capability of Edge servers. In the CC, heterogeneous Cloud and Edge Servers together support the deployment of diverse applications such as emerging ML services. As ML services require both quick responsiveness and high computational efficiency, service migration across CC has become important to balance the service performance, cost, power, and energy efficiency [24].

A basis enabler of the service migration in the CC is containerization, because it provides a lightweight, portable and isolated execution environment for ML services. Using container orchestrators such as Docker Swarm[6], Mesos[8], and Kubernetes[11], a containerized ML service can be flexibly deployed and scheduled within the CC. Regarding resource management, Kubernetes can support some autonomous management, such as ML service self-healing and ML service auto-scaling. However, concerning service migration, current orchestrators support service restarting and assigning target nodes for service migration [25], but providing the capability to dynamically move the ML service between Cloud and Edge servers based on real-time system or application uncertainty is still challenging.

ML techniques can be used for decision-making under uncertainty. For instance, Reinforcement Learning (RL) is able to learn the optimal behavior as a policy by continuously interacting with the environment [3]. Some works utilize RL for service placement [15, 21], while others provide RL-driven auto-scaling solutions [20, 22]. Only a few consider RL to manage container migrations [23]. Moreover, those works mostly focus on optimizing metrics such as latency and resource utilization. However, for current ML services that process a model, computation-intensive processing requires consideration of power/energy efficiency.

In this paper, we look for an intelligent migration methodology for migrating containerized ML applications in the CC. The goal is to continuously find the performance-power optimal deployment of the ML application at runtime under a dynamic environment. Therefore, we present Greenwise framework, in which our main contributions are:

- We present real-time power and performance monitoring at both node- and container-level. The monitoring agent contains the capability to adapt power models for heterogeneous Cloud and Edge nodes.
- We study an RL-based approach for ML application migration within the CC infrastructure, with a balanced performance-power optimal objective and also continuous training in an online manner to adapt to dynamic environments.
- We establish a Kubernetes-based CC platform, implement the monitoring, planning and execution, and evaluate the

containerized ML service online migration with the full GreenWise framework.

2 Background and Related Work

2.1 Service Migration in the CC

Service migration has become an essential operation in the CC environment. This operation can relocate the running services seamlessly from one host to another to ensure continuity, efficient resource utilization and application performance or other system metrics. Traditional virtualization technologies provided the way for live migration, allowing virtual machines (VMs) to move among hosts without significant downtime or loss of state [12]. However, in the current CC, containerization and container orchestration platforms have been widely utilized so that the service migration has evolved.

Guitart [7] explores the use of CRIU's advanced features to implement diskless, iterative (pre-copy and post-copy) migrations of containers with external network namespaces and established TCP connections, so that HPC applications can live-migrate. This work demonstrated that there is a need to integrate Checkpoint/Restore with containerization, and live-migration is feasible and practicable when properly configured.

Kubernetes has emerged as the dominant orchestration tool for containerized applications, thus there is work focusing on seamless service migration on Kubernetes. Natively, service migration on Kubernetes is done by stopping the pod and redeploying it. This is considered an acceptable way for migrating a stateless service. Other works address stateful service migration. Vasireddy et al. [25] proposed a method optimizing pod migration by effectively managing the persistent volumes in a Kubernetes environment. Zhang et al. [27] introduced KubeSPT, which synchronizes the network state of pods and internal containers by controlling packet flow and implements fast service redirection. Also, they introduced a Hot Data and Lazy-Restore method for memory restoration to reduce migration downtime. Works [10, 18] consider more complex migration scenarios. Junior et al. [10] explored seamless pod migrations in geo-distributed Kubernetes clusters, and Poggiani et al. [18] investigated multi-container pod migrations, demonstrating the need to coordinate migration strategies.

These techniques provide diverse strategies for migrating services within CC. However, considering our ML service - a serving platform that isolates model inference processes without maintaining internal state between requests - is stateless, we leverage Kubernetes' native capability for ML service migration through efficient pod restarting, where we properly patch the service reallocation and ensure a graceful shutdown time.

2.2 RL-based Approaches for Container Migration Management

ML techniques can be used for migration, such as supervised learning, deep learning (DL), RL/deep reinforcement learning (DRL), or hybrid approaches [3]. Among these, RL is a self-learning technique in which no prior knowledge of the environment is needed. The agent learns the optimal behaviour by continuously interacting with the environment.

In addition, other works [15, 21] utilize RL for service placement. The authors schedule complex microservices-based applications on Kubernetes within a CC or 6G environment. Others focus on management, which provides RL-driven auto-scaling solutions [20, 22], where the primary objective of these works is to build a model for dynamically scaling resources based on real-time demand. Moreover, these works focus mostly on optimizing metrics such as latency and resource utilization. However, for current ML services that process a model, computation-intensive processing requires consideration of power/energy efficiency.

Only a few consider RL to manage container migrations. Tang et al. [23] proposed DRL-based container migration algorithms in a fog environment to support mobility tasks with various application requirements. Liu et al. [12] provided a hierarchical framework combining RL, auto-encoder and Long short-term memory network (LSTM) to optimize the resource allocation and power consumption in the Cloud. Ren et al. [19] trained a DRL model using Proximal Policy Optimization (PPO) while using an LSTM for policy network to extract time-series features. However, these works target the Cloud for VMs migrations.

2.3 Power Monitoring in the CC

ML services are computation-intensive applications which have an increasing demand for concerning their power efficiency. Power monitoring is a foundational step toward achieving energy-efficient and sustainable CC.

Intel's Running Average Power Limit (RAPL) interface allows to precisely measure energy consumption at the host and process levels. However, to work with today's containerization and container orchestrator, some tools are needed to provide power consumption measurement at container-level. Scaphandre [16] leverages hardware-based sensors RAPL to measure power consumption and has Kubernetes support. Kepler [2], an open-source cloud-native tool led by Red Hat, IBM, and Intel, uniquely combines software counters with hardware power sources (e.g., RAPL, Advanced Configuration and Power Interface (ACPI), NVIDIA GPUs, extended Berkeley Packet Filter (e-BPF)), offering granular, pod-specific power consumption metrics within Kubernetes clusters. Additionally, Kepler employs machine learning models to enhance power estimations. KubeWatt [17] provides a prototype for demonstrating higher accuracy for container-level energy usage than Kepler. However, it relies on a specific Redfish API to get power measurement data.

[4] provided a platform- and application-agnostic methodology for full-system power modeling in heterogeneous data centers. The model has high accuracy with different patterns of resource usage and energy consumption, by systematically selecting a minimum set of indicators. [5] analyzed open-source power measurement tools (i.e., S-tui, Kepler, and Scaphandre) applicable to containerized environments. [1] presents a monitoring framework and shows Scaphandre's underestimation of memory impact on power consumption and Kepler's limitation in host metrics.

3 GreenWise Framework

We first introduce the service migration scenario in the CC, and then provide a system architecture of the GreenWise framework with a Monitoring-Planning-Execution loop to achieve online dynamic service migration.

3.1 Problem Statement

Figure 1 shows a potential ML service migration scenario. In the infrastructure layer, Cloud and Edge servers within the CC can provide resources for ML services. Those servers are heterogeneous and with different power profiles and changing available resources. In the application layer, the ML service will receive fluctuating user requests. Given these dynamic system state changes and workload fluctuation, it is critical to design intelligent migration strategies that dynamically adapt ML service deployment across the continuum in response to the changes, in order to achieve the performance and power efficiency goals.

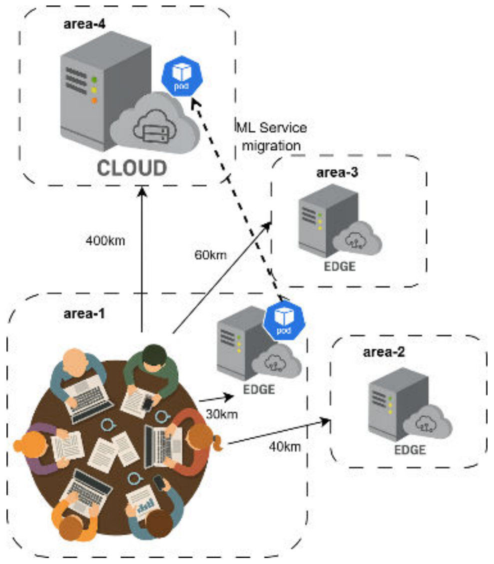

Figure 1: Migration of ML Applications in the Computing Continuum.

The main objective is twofold:

- Propose a decision-making framework that enables efficient and autonomous migration of ML services, by developing an agent that interacts with the environment to monitor the power consumption and performance of hosts and applications across the CC, to employ online RL training and to enable efficient migration of ML services within the CC. See Section 3.2.
- Formulate the service migration problem as a Markov Decision Process (MDP), leveraging RL techniques to observe current states of systems and applications, select optimal migration actions (i.e., choosing the host for service migration), and optimize multiple objectives—such as minimizing power consumption and latency—to maximize rewards. Further details are in Section 4.

3.2 GreenWise System Architecture

Figure 2 shows the Kubernetes-based CC platform. The GreenWise framework is shown as the orange blocks, which implement Monitoring-Planning-Execution as a dynamic decision-making loop. All components are currently working in a cloud-native manner, however, if the container orchestration platform changes, the monitoring and execution blocks can be adapted to the new platform as well.

Power and Performance Monitoring: The monitoring stage occurs on each host and its objective is to provide comprehensive monitoring of the power and performance (i.e., utilization) of the host, and power and performance (i.e., latency, throughput, etc.) of containers running on this host. Table 1 shows the source of those metrics. Specifically, the resource usage and power metrics are collected by Kepler [2], where Kepler gets the real-time CPU count from the system eBPF and power metrics from dynamic power model prediction and idle power fair-sharing. On the other hand, the application QoS metrics and the workload user monitoring are done by TorchServe and custom exporters, respectively. All metrics are stored in Prometheus[1], which is then available to be consumed by the planner agent, enabling decisions based on up-to-date observations.

Table 1: Sources of Host and Container Metrics

Metric Type	Metric	Source
Host States	CPU Usage	Kepler Exporter
	Power Consumption	Kepler Exporter
App. States	Latency	TorchServe Runtime Exporter
	Throughput	TorchServe Runtime Exporter
	Number of Users	Client Exporter
	CPU Usage	Kepler Exporter
	Power Consumption	Kepler Exporter

RL-based Planning Agent: The planning stage is an online agent that continuously interacts with the environment and makes migration decisions in real-time. On the one hand, it connects with Prometheus to get the monitoring data for its observation and the container orchestrator Kubernetes to enable the migration operation. On the other hand, it uses RL methodology to online learn the best migration policy concerning power-performance objectives within the CC. The agent is built and extended based on the DataConnector[13] and Gwydion[20]. The migration problem is modeled as an MDP, and more details will be explained in Section 4.

Migration Execution: The migration execution is the last stage that allows the migration actions within the CC environment. As discussed in Section 2.1, for stateless ML services, the native Kubernetes capability can provide efficient ML service migration by pod restarting. When there is a migration decision that specifies the current and target hosts, the migration executor verifies the feasibility of the migration. If the target host can accommodate the service, the executor patches the pod's allocation accordingly to initiate the migration.

4 Power-Performance Aware ML Service Migration

In this section, we present the application power and performance model, and then we introduce an RL approach for service migration,

[1]https://prometheus.io/

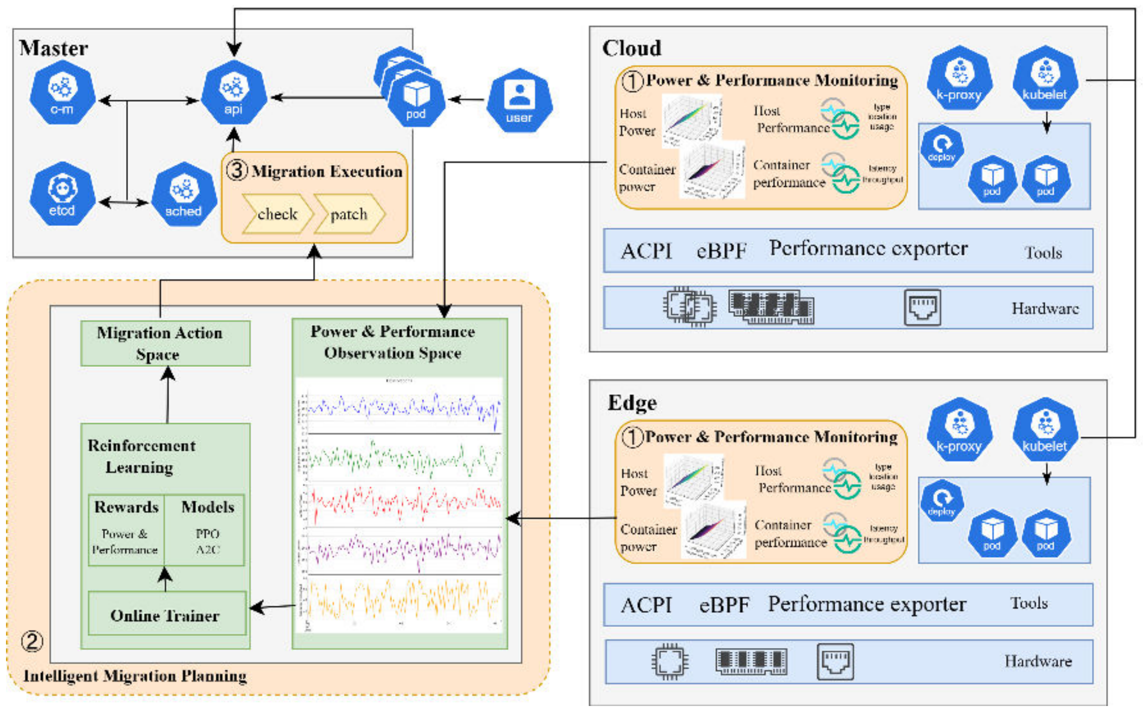

Figure 2: GreenWise: Kubernetes-based Computing Continuum Platform for Intelligent Migrating ML Services.

formulating it in detail as an MDP by clearly defining states, actions, and the reward function. Notations are listed in Table 2.

4.1 Application Power and Performance Model

4.1.1 Power model. The power model of the application quantifies the energy usage of the application a on the host h at time t. It is composed of idle power and dynamic power as shown in Equation 1. Specifically, idle power consumption, denoted $P\_idle^t_{a,h}$, is calculated proportionally based on the fraction of processes assigned to the application over the total processes on the host (as shown in Equation 2). This ensures that idle energy is fairly distributed between applications that share the same host. The dynamic power corresponds to the energy consumed by active computation, which is influenced by different user demands as well as the characteristics of different power models of heterogeneous hosts. We have enabled different power models in the Kepler exporter agent for estimating the power for heterogeneous hosts based on the CPU usage.

$$P^t_{a,h} = P\_dyna^t_{a,h} + P\_idle^t_{a,h} \tag{1}$$

$$P\_idle^t_{a,h} = \frac{N\_process^t_{a,h}}{N\_process^t_h} * P\_idle_h \tag{2}$$

4.1.2 Performance model. The application performance model presents the end-to-end latency $L^t_{a,h}$ of application a running on host h at time t. This total latency is a sum of computation and transmission delays, which are essential for ML services (as shown in Equation 3). The backend latency $(L\_back^t_{a,h})$ is the processing latency within the service backend, and the network latency $(L\_net_{a,h})$ is the communication delay between the users and the host that allocates the application.

$$L^t_{a,h} = L\_back^t_{a,h} + L\_net_{a,h} \tag{3}$$

4.2 Markov Decision Process Formulation

The allocation of the ML service finally decides the quality of service (QoS). However, the migration decision is affected by several variables that change over time, such as the current host states and workloads. Thus, we focus on using RL for service-level migration to find the most suitable host at time t for application a, which optimizes the objectives of minimizing application power and maximizing QoS. We model the service migration process as an MDP.

Markov Decision Process Definition: An MDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where:

- $\mathcal{S}$: finite set of states,
- $\mathcal{A}$: finite set of actions,
- $\mathcal{P}(s' \mid s, a)$: transition probability function,
- $\mathcal{R}(s, a, s')$: reward function,
- γ: discount factor $(0 \leq \gamma \leq 1)$.

4.2.1 State. $\mathcal{S}$ is the state space. The agent is expected to construct state s_t by periodically checking the current ML service placement and collecting the information from the CC monitoring stack. The full state s_t consists of state tuples in both all hosts states $\mathbf{s}^t$ and application states s^t_a, expressed in Equation 4.

$$s_t = \langle s^t_a, \mathbf{s}^t \rangle \quad s_t \in \mathcal{S} \tag{4}$$

Table 2: Summary of Notations

Notation	Description
N	Number of hosts
h	Host number selected for app. deployment
$\mathcal{S}$	State space
s_t	State at time t, $s_t = \langle s_a^t, \mathbf{s}^t \rangle$
s_a^t	App. state tuple at time t
$\mathbf{s}^t$	All hosts states at time t
s_h^t	Host-specific state tuple at time t
C_h^t	CPU utilization of host h at time t
$C_{a,h}^t$	CPU utilization of app. a on host h at time t
$N\_process_h^t$	Number of processes on host h at time t
$N\_process_{a,h}^t$	Number of processes belonging to app. a on host h at time t
P_h^t	Power consumption of host h at time t
P_max_h	Maximum power capacity of host h
P_idle_h	Idle power consumption of host h
$P_{a,h}^t$	Power consumption of app. a on host h at time t
$P\_dyna_{a,h}^t$	Dynamic power consumption of app. a on host h at time t
$P\_idle_{a,h}^t$	Idle power consumption of app. a on host h at time t
$P_{a,h}^t$	Power consumption of app. a on host h at time t
$L_{a,h}^t$	End-to-end latency of requesting app. a on host h at time t
$L\_net_{a,h}$	Network latency from client to host h
$L\_back_{a,h}^t$	Backend latency of app. a on host h at time t
L_max_a	Maximum latency of requesting app. a
U_a^t	Number of users requesting app. a at time t
SLA_a	SLA constraint for app. a
$\mathcal{A}$	Action space
a_t	Action (app. migration decision) at time t
$mask_t$	Action mask to filter invalid host at time t
$\mathcal{R}(s, a, s')$	Reward function
r_t	Reward at time t
r_P^t	Power efficiency reward at time t
r_{QoS}^t	QoS reward (latency) at time t
$\mathcal{P}(s' \mid s, a)$	Transition probability function
$p(s_t' \mid s_t, a_t)$	Probability of moving to s_t' after taking a_t in s_t
γ	Discount factor for future rewards

$$\mathbf{s}^t = \{s_1^t, s_2^t, \ldots, s_N^t\}, \quad \text{where} \quad s_h^t = \langle P_h^t, C_h^t \rangle \tag{5}$$

$\mathbf{s}^t$ denotes the states of all hosts at time t, as shown in Equation 5. Each element s_h^t specifies a state tuple of host h, where P_h^t represents the current power consumption of host h. C_h^t represents the resource utilization (*e.g.*, CPUs) of host h.

$$s_a^t = \langle h, P_{a,h}^t, C_{a,h}^t, L_{a,h}^t, U_a^t \rangle \tag{6}$$

Equation 6 defines the state of application a at time t, which contains the application deployment location and also power and performance metrics. The state is represented as a tuple, where:

- h is the index of the host on which the application a is currently deployed.
- $C_{a,h}^t$ is the application a CPU resource usage.
- U_a^t is the number of users requesting application a.
- $P_{a,h}^t$ is the current observed power consumption of the application a deployed on host h.
- $L_{a,h}^t$ is the end-to-end latency observed in the application a deployed on host h.

4.2.2 Action. A is the action space. The action of the agent is to find the migration placement of an application, so the action space is the migration of the application to the target host h. To guide the agent away from invalid actions, a simple approach is to penalize the agent if an invalid action is selected. Another efficient way in the literature is to use action masking [9, 21], which can prefilter the valid actions based on the current state so the agent can know it beforehand. This approach has recently shown significantly higher performance and sample efficiency than penalties. The action masks for each host h in state can be defined as Equation 7, meaning migrating to the current host is invalid.

$$mask_t = \begin{cases} \text{false} & \text{If the host } h \text{ is the current host,} \\ \text{true} & \text{Otherwise.} \end{cases} \tag{7}$$

At the scheduling time t, the action can be expressed as Equation 8, the agent can choose no migration or migrate to a specific host h.

$$a_t = \begin{cases} 0 & \text{no action} \\ h & h \in \{1, 2, \ldots, N\} \end{cases}, a_t \in \mathcal{A} \tag{8}$$

4.2.3 Reward. $\mathcal{R}(s, a, s')$ is the reward function. The agent aims to maximize the accumulated reward $\sum_{t=0}^{T} \gamma^t r_t$, where r_t is the reward when the agent acts the service migration at each scheduling time t. In the CC in which the ML service is deployed, the objective is to minimize application power consumption and maximize application QoS (i.e., minimize latency).

Equation 9 and Equation 10 show the normalized reward for the power consumption and the end-to-end latency, respectively.

$$r\_P_{a,h}^t = \alpha(1 - \frac{P_{a,h}^t}{P\_max}) \tag{9}$$

$$r\_QoS_{a,h}^t = \beta(1 - \frac{L_{a,h}^t}{L\_max}) \tag{10}$$

The total reward is a combination of both rewards, meaning considering a balanced power and performance for the service migration, (i.e., *Power-Latency Aware*). If an SLA violation occurs that reaches the maximum application latency, a higher penalty of -1 is applied (as shown in Equation 11).

$$r_t(s_t, a_t) = \begin{cases} -1 & L_{a,h}^t > L_{max} \\ r\_P_{a,h}^t + r\_QoS_{a,h}^t & otherwise \end{cases} \tag{11}$$

For comparison, we also model different rewards for the agent, namely *Power-Aware* where the reward minimizes the power usage only, represented as $r_t(s_t, a_t) = r\_QoS_{a,h}^t(\beta = 1)$, and *Latency-Aware* where the reward minimizes the latency only, represented as, $r_t(s_t, a_t) = \begin{cases} -1 & L_{a,h}^t > L_{max} \\ r\_P_{a,h}^t(\alpha = 1) & otherwise \end{cases}$.

4.2.4 Transition Probability. $\mathcal{P}(s' \mid s, a)$ is the the state transition function. The transition probability at time t $p(s_t' \mid s_t, a_t)$ indicates current state s_t transiting to next state s_t' after taking action a_t, and satisfy $\sum_{s_t' \in \mathcal{S}} p(s_t' \mid s_t, a_t) = 1$.

Table 3: Host Specifications and Power Consumption

[Host]Processor	Cores	Chips	Memory (GiB)	Avg Max Watt	Avg Idle Watt
[cloud] Intel Xeon E5-2620	12.0	2.0	128.0	280.0	162.0
[edge0] Intel Xeon E3-1265lv3	4.0	1.0	8.0	58.5	19.0
[edge1] Intel Xeon E3-1220v2	4.0	1.0	8.0	69.7	36.6
[edge2] Intel Xeon E3-1270	4.0	1.0	8.0	102	21.4

Table 4: Client to Host Distance and Network Latency

[Host]Area (Client located at 'area-1')	Distance(km)	Network Latency(ms)
[edge2] area-1	30km	3ms
[edge1] area-2	40km	5ms
[edge0] area-3	60km	14ms
[cloud] area-4	400km	30.9ms

5 Evaluation

In this section, we first introduce the experimental setup. In the experiments, we evaluate the power models for monitoring, RL models for service migrations planning, and finally we test the GreenWise for intelligent service migration in a real cluster mode.

5.1 Experimental Settings

Platform Settings: Our experiments are executed on a four-node Kubernetes cluster, where the control-plane consists of 12 cores, 48GiB RAM; three edge hosts of 4 cores, 8GiB RAM. Each host is an Openstack-based VM with Rocky Linux 9.3. The CC platform is built based on Kubernetes v1.28.2 (Network: Calico v3.26.4, Runtime: containerd v1.6.25, DNS: CoreDNS v1.10.1, Storage: etcd 3.5.9).

Application Settings: We use a target ML-serving service application for image classification. The application is running a typical ML model (Resnet_18), wrapped within TorchServe[2] and deployed as a Kubernetes deployment. The serving is set with client_threads=32, job_queue_size=1000, max_batch_delay=100, batch_size=8 and inter/intra operation threads=1 to make the service efficiently use all CPU resources.

Client Settings: We use the Locust[3] client to generate users (up to 60) sending images with a random waiting time between 0.1 and 0.5 seconds to the TorchServe ML service and receiving image classification results. A customer exporter is loaded to monitor the number of user changes. For training, we use an incremental client pattern from 1 to 60 and loop it to collect data and do online training; for testing, we use a new environment that uses a client pattern close to a Gaussian distribution from a real dataset [4].

Emulate Settings: We emulated different profiles for hosts and also the distance and network latency between the client and hosts.

- Host Power Profile: As our hosts are VMs that do not have ACPI or RAPL mounted to report platform or component energy consumption, we emulate the four heterogeneous VMs using different host profiles as shown in Table 3, and enable power estimation for those hosts using Kepler.
- Client to Host Distance and Network Latency: We assume clients are in area-1, and the distance between those clients and hosts in different areas within the CC is shown in Figure 1. Work [26] collected real-world datasets on latency and analyzed the impact of distance on latency, thus, we took their empirical results and models for estimating the network latency of client to host with various distances (as shown in Table 4).

[2]https://github.com/pytorch/serve
[3]https://github.com/locustio/locust
[4]https://github.com/Azure/AzurePublicDataset

Monitoring Settings: Kepler [2][5] v7.8.0 is deployed on each host for calculating host and pod power consumption; TorchServe is configured to measure the application performance. All Kepler, TorchServe and Locust internal metrics are exported to Prometheus at every monitoring interval (i.e., 3s or 15s), and the Prometheus internal defined rules synchronously aggregate GreenWise useful power and performance data by host and pod every 10s. Table 5 shows all monitoring settings. Note that Kepler is still evolving that each version may differ. We use v7.8.0 and have patched it with heterogeneous power model prediction, have enabled idle power estimation and have also fixed a unit bug that reports kilowatt instead of watt, which confused the prediction model.

Table 5: Monitoring Configuration

Sources	Config	Value
Kepler	ENABLE_EBPF_CGROUPID	true
	ENABLE_PROCESS_METRICS	true
	EXPOSE_ESTIMATED_IDLE _POWER_METRICS	true
	MODEL_CONFIG: NODE_TOTAL_ESTIMATOR	true
	PROMETHEUS_SCRAPE_INTERVAL	3s
TorchServe	metrics.enabled	true
	metrics_mode	prometheus
	metrics.flush_interval	3000ms
Prometheus	Kepler ServiceMonitor interval	3s
	Kepler PrometheusRule interval	10s
	TorchServe ServiceMonitor interval	3s
	TorchServe PrometheusRule interval	10s
	Locust ServiceMonitor interval	15s

Agent Settings: The GreenWise framework has been implemented in Python using OpenAI Gym and stable baselines 3 library. The autonomous loop is based on [13] and the basic Gym environment is based on [20]. We have enabled the previous MDP model within the agent, connecting observations retrieved from

[5]https://github.com/sustainable-computing-io/kepler

Prometheus and also triggering the migration actions by Kubernetes API. The online Trainer we realized a power performance balanced objective and trained the model with different popular DRL algorithms, such as Masked Proximal Policy Optimization (MaskPPO) [9] and Advantage Actor Critic (A2C) [14]. In the evaluation, we trained the agent for 5500 total steps and tested it for 1000 total steps. Each episode consists of 25 steps. Note that the collected data for training and testing is also used for training and testing other models in simulation for comparison purposes.

Migration Strategies Settings: Table 6 shows the migration strategies and algorithms we use for ML service migration. *Random* and *Round-Robin* are the baseline strategies with simple randomized/cycled host selections; *Power-Aware* and *Latency-Aware* are single-objective strategies focusing only on minimizing power and minimizing latency, respectively. *Power-Latency-Aware* is our proposed strategy that balances power and latency objectives ($\alpha = 0.5$ and $\beta = 0.5$), details have been shown in Section 3. Moreover, we use the MaskPPO algorithm to train agents for different objectives (i.e., *Power-Aware*, *Latency-Aware*, and *Power-Latency-Aware*), and for *Power-Latency-Aware* strategy for GreenWise, we also trained the agent with different algorithms (i.e., A2C) or with different environments (i.e., cluster).

Table 6: Migration Strategies Settings

Strategies	Algorithms	Notes
Random	R	Randomized hosts
Round-Robin	RR	Cycle evenly across hosts
Power-Aware	MaskPPO	Prioritize power efficiency
Latency-Aware	MaskPPO	Prioritize low latency
Power-Latency-Aware (GreenWise)	MaskPPO	Balance power and latency
	A2C	Balance power and latency
	MaskPPO-cluster	Balance power and latency (train/eval. in a real cluster)

5.2 Host and Pod Power Monitoring Evaluation

In the first experiment, we evaluate a set of power models for heterogeneous hosts. This allows GreenWise to utilize the most accurate models to monitor the power consumption of different types of hosts. The power model is trained with different ML algorithms using scikit-learn based on the SPEC power [6] data. Focusing on computation-intensive ML services, the power model selects CPU usage (i.e., bpf_cpu_count) as the key feature that contributes to the power. These ML models are evaluated by using the following metrics:

- **Mean Absolute Error (MAE):** computes the average of the absolute differences between predicted and actual values, treating all errors equally.
- **Mean Absolute Percentage Error (MAPE):** average percentage difference between predicted and actual values, providing insights into the relative error.

Table 7 and Figure 3 show the comparison of MAE and MAPE of ML models for each host. For cloud, edge0 and edge1, the Polynomial Regression model shows the best MAE/MAPE, and for edge2, the GradientBoosting Regression model shows the best accuracy. We have enhanced the Kepler exporter agent for GreenWise monitoring, and added support to Kepler to use different power models for these heterogeneous hosts.

[6]https://www.spec.org/power_ssj2008/results/

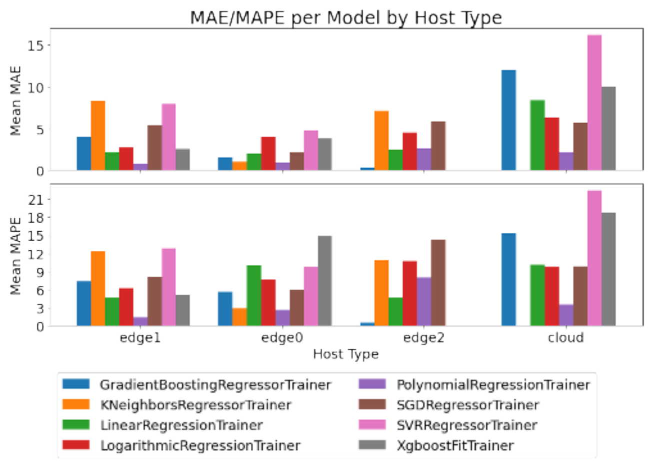

Figure 3: Machine Learning based power models for different hosts.

5.3 RL Models for Migration Evaluation

In the second experiment, we evaluate different RL models for service migrations in simulation. First, we analyze the results of training and testing RL models using different migration strategies and different algorithms. After that, we compare service migration using baselines and RL-based GreenWise strategies.

RL Models Training: We run online training using *Power-Latency-Aware* strategy and MaskPPO algorithm on the Kubernetes-based CC platform for ML service migration. The data generated from the real cluster is used as the simulation dataset for other RL model training. The performance of the model is evaluated by the following metrics:

- **Accumulated reward:** It refers to the total sum of rewards per episode obtained by an agent over time as it interacts with the environment (max 25).
- **Average power consumption:** power consumed by the application during each episode.
- **Average application latency:** end-to-end application response time during each episode.

Figure 4 illustrates the CDF (Cumulative Distribution Function) for *Power-Aware*, *Latency-Aware* and *Power-Latency-Aware* strategies during training. For the *Power-Latency-Aware* strategy, it also presents different RL algorithms such as MaskPPO and A2C. As shown in the results, all MaskPPO variants in simulation mode outperform A2C with steeper curves (low variance and more predictable behavior) and better median rewards, meaning MaskPPO provides more stable training. However, the MaskPPO-cluster mode shows a shift from the MaskPPO simulation mode and shows worse tail latency/power. This can be related to unstable API calls or metrics retrieval retry from Prometheus. Also, it can be attributed to OS jitter and burstier co-tenancy spikes in the real cluster training, which result in fatter tails for power and latency.

Table 7: Comparison of MAE and MAPE of ML Models for Each Host

Host Type / ML Models	cloud		edge0		edge1		edge2	
	MAE	MAPE	MAE	MAPE	MAE	MAPE	MAE	MAPE
GradientBoostingRegressor	12.0	15.3	1.52	5.62	4.0	7.4	**0.27**	**0.6**
KNeighborsRegressor	—	—	1.03	2.95	8.3	12.31	7.08	10.85
LinearRegression	8.38	10.12	2.03	9.97	2.2	4.66	2.42	4.68
LogarithmicRegression	6.33	9.77	4.00	7.67	2.75	6.29	4.55	10.73
PolynomialRegression	**2.23**	**3.45**	**0.95**	**2.71**	**0.76**	**1.43**	2.61	8.02
SGDRegressor	5.67	9.8	5.91	6.02	5.46	8.16	5.91	14.21
SVRRegressor	16.15	22.34	4.80	9.72	7.96	12.83	—	—
XgboostFit	9.99	18.64	3.80	14.76	2.53	5.1	—	—

Note: The best values for MAE and MAPE in each column are highlighted in **bold**. Values over than threshold 25 are represented by "—".

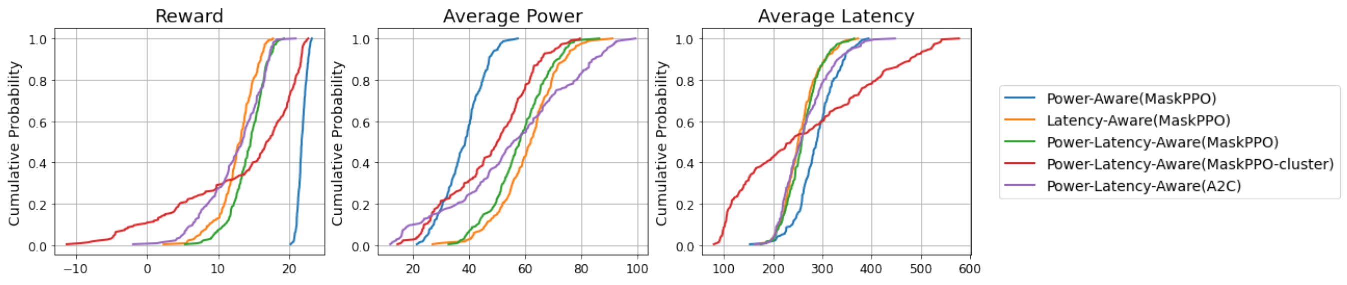

Figure 4: The training results for different strategies and algorithms (CDF)

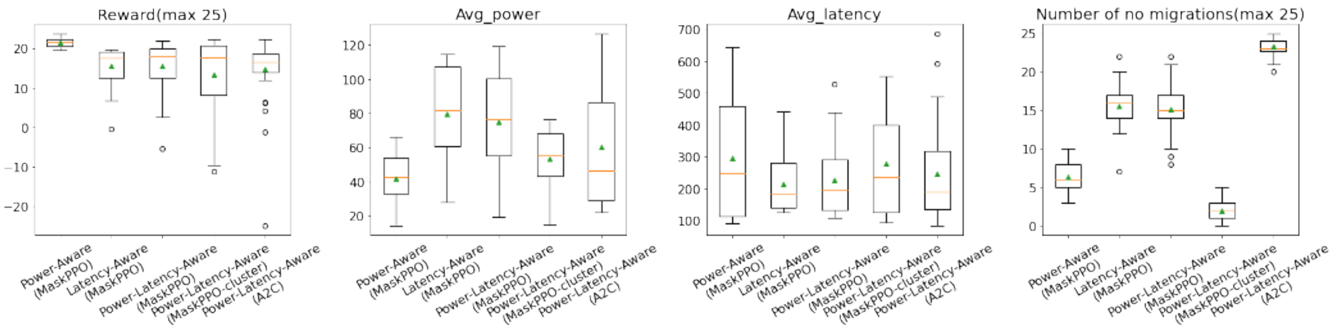

Figure 5: The testing results for different strategies and algorithms

RL Models Testing: After training these DRL models with different strategies and algorithms, we evaluate these agents with a different user demand (Gaussian distribution generated by Azure dataset) pattern. Figure 5 shows the distribution of different metrics over 40 episodes during testing. Comparing the different strategies, the results show that the *Power-Aware(MaskPPO)*, focusing on the power efficiency, has the lowest average power and highest average latency, and the *Latency-Aware(MaskPPO)*, prioritizing low latency, has the lowest average latency and highest average power. The *Power-Latency-Aware(MaskPPO, MaskPPO-cluster, A2C)* considers a balanced power and latency, so that its average power and average latency sit between the *Power-Aware* and *Latency-Aware* agents, thus showing GreenWise *Power-Latency-Aware* strategy has a better trade-off between power and latency.

Service Migration Comparison: After training and testing different RL models, we then compare two baseline migration strategies (*Random*, *Round-Robin*) with the GreenWise RL agent that uses strategies *Power-Latency-Aware(MaskPPO, MaskPPO-cluster, A2C)*. Figure 6 shows the service migration simulation results over 10 runs. Both baselines show lower rewards than the RL-based agents, i.e., the best RL agent *Power-Latency-Aware(MaskPPO)* is 206.28% better than *Random* and 29.77% better than *Round-Robin* in mean. *Random* and *Round-Robin* have similar power and latency metrics, but *Random* achieves lower rewards because *Random* can suffer a penalized action of migrating to the same host, which does not

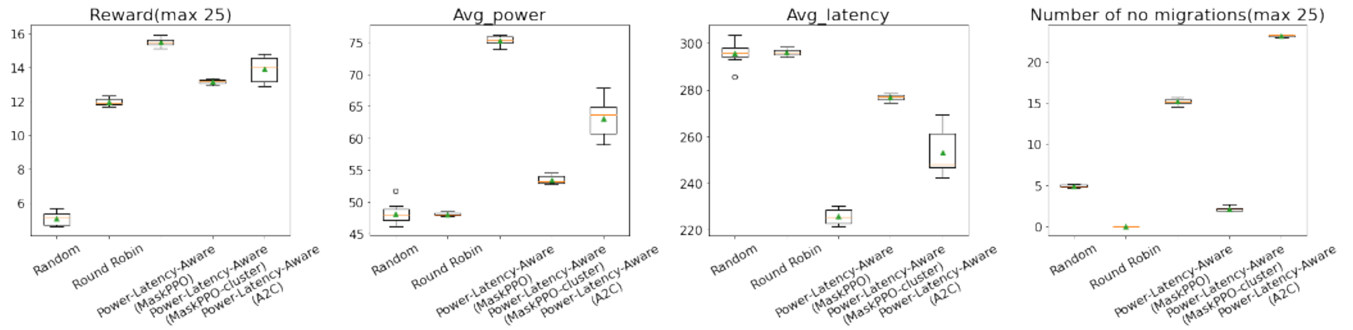

Figure 6: Service migration simulation results over 10 runs

happen in *Round-Robin*. Also, all RL agents will avoid this penalty because migrating to the same host action is masked while training.

Comparing different RL algorithms for *Power-Latency-Aware* strategy within the GreenWise. *Power-Latency-Aware (MaskPPO)* achieves the highest reward. *Power-Latency-Aware (MaskPPO-cluster)* and *Power-Latency-Aware(A2C)* follow, showing competitive rewards but slightly lower than *Power-Latency-Aware (MaskPPO)*. Comparing the power and latency metrics, different agents show unique trade-offs. *Power-Latency-Aware (MaskPPO)* minimizes latency the best but requires more power, and *Power-Latency-Aware (MaskPPO-cluster)* achieves lower power consumption but at a higher latency. *Power-Latency-Aware (A2C)* stays in the middle between *Power-Latency-Aware (MaskPPO, MaskPPO-cluster)* but with greater variance. This is because with different initial conditions among different runs, it converges to different local optima host and performs no migrations (as shown in the number of no migrations), leading to variability across runs. Overall, *Power-Latency-Aware (MaskPPO)* has the best efficiency, and *Power-Latency-Aware (MaskPPO-cluster)* is less efficient but suitable for power-sensitive scenarios, thus making both of them good candidates for real cluster leverage.

5.4 GreenWise Service Migration Evaluation

Finally, we evaluate the effectiveness of the full GreenWise framework for intelligent service migration in a real cluster. We compare GreenWise integrated agents (i.e, *Power-Latency-Aware (MaskPPO or MaskPPO-cluster)*) with two baselines (i.e., *Random*, *Round-Robin*) to advise the service migration with the same execution time. Table 8 presents the results of different migration strategies. It shows that the *Power-Latency-Aware (MaskPPO)* agent has the best efficiency, as its reward is 159.2% better than the *Random* and 13.8% better than the *Round-Robin* baselines. This is slightly worse than the simulation results, because in the real cluster, we identified a slight drift in both overall power consumption and latency when compared to the simulation dataset (see table 8 and also figure 6). In general, the system is less busy than the dataset we collect for simulation.

Figure 7 to Figure 9 show more details of the execution using different migration strategies. Figures 7 and 8 show the natural variance in the real cluster in both power consumption and latency, which is expected due to changing workload and also the state of the system. Despite this variance, *Power-Latency-Aware*

Table 8: Average Metrics Across Migration Strategies

	Random	Round-Robin	Power-Latency-Aware	
			MaskPPO	MaskPPO-cluster
avg_reward	7.703	17.545	19.966	18.670
avg_power	38.936	35.866	52.490	39.476
avg_latency	191.695	202.827	131.209	187.701

(MaskPPO) achieves the lowest latency in gaining rewards under dynamic conditions. Figure 9 shows the time distribution of the application running on each host. The results show that the *Random* baseline application running time is close to *Round-Robin*, where the application runs evenly distributed across hosts. However, the DRL agents within GreenWise are more intelligent, where *Power-Latency-Aware (MaskPPO-cluster)* prefers the cloud and the edge 0 hosts, and *Power-Latency-Aware (MaskPPO)* indicates a more stable policy preference to the cloud host.

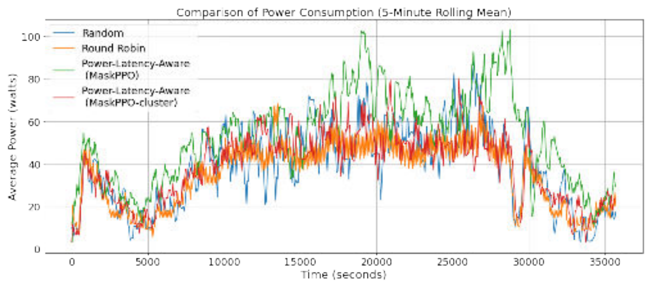

Figure 7: The average power trend on a real cluster

6 Conclusion

This paper presented the GreenWise framework for intelligently migrating containerized ML services in a Kubernetes-based CC platform. We extended the monitoring agents with heterogeneous power models, and trained and utilized different migration strategies with different intelligent algorithms in the planner agent for sustainable practices. Our results show that the proposed GreenWise with a Power-Latency-Aware MaskPPO agent can outperform Random or Round-Robin by 159.2% and 13.8% for service migration in a real cluster. In the future, we plan to integrate drift detection

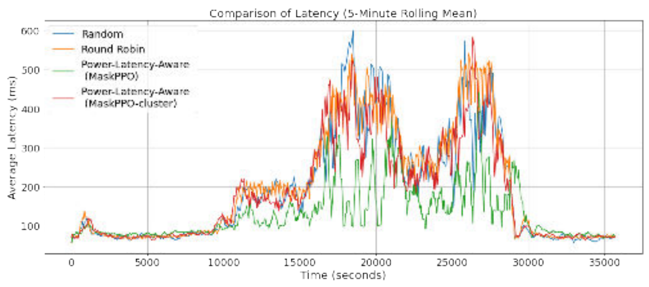

Figure 8: The average latency trend on a real cluster

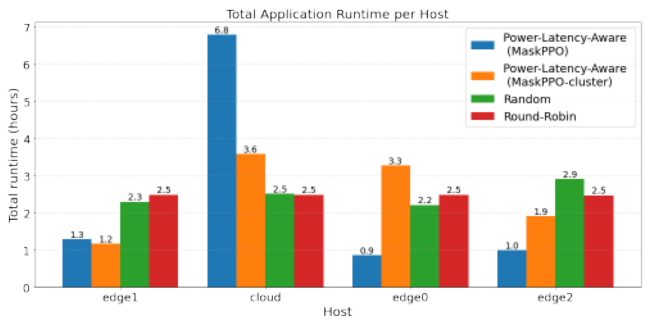

Figure 9: The application runtime distribution on a real cluster

mechanisms to adapt policies when workload or system behavior shifts over time. We also aim to enhance fine-grain monitoring for migration process and design models that explicitly capture and predict migration costs, enabling more precise decision-making under dynamic conditions.

References

[1] Milad Akbari, Raffaele Bolla, Roberto Bruschi, Franco Davoli, Chiara Lombardo, and Beatrice Siccardi. 2024. A Monitoring, Observability and Analytics Framework to Improve the Sustainability of B5G Technologies. In *2024 IEEE International Conference on Communications Workshops (ICC Workshops)*. 969–975. doi:10.1109/ICCWorkshops59551.2024.10615948

[2] Marcelo Amaral, Huamin Chen, Tatsuhiro Chiba, Rina Nakazawa, Sunyanan Choochotkaew, Eun Kyung Lee, and Tamar Eilam. 2023. Kepler: A Framework to Calculate the Energy Consumption of Containerized Applications. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. 69–71. doi:10.1109/CLOUD60044.2023.00017

[3] Nour El Houda Boubaker, Karim Zarour, Nawal Guermouche, and Djamel Benmerzoug. 2022. Fog and Edge Service Migration Approaches based on Machine Learning Techniques: A Short Survey.. In *TACC*. 33–44.

[4] Mauro Canuto, Raimon Bosch, Mario Macias, and Jordi Guitart. 2016. A methodology for full-system power modeling in heterogeneous data centers. In *Proceedings of the 9th International Conference on Utility and Cloud Computing* (Shanghai, China) *(UCC '16)*. Association for Computing Machinery, New York, NY, USA, 20–29. doi:10.1145/2996890.2996899

[5] Centofanti, Carlo and Santos, José and Gudepu, Venkateswarlu and Kondepu, Koteswararao. 2024. Impact of power consumption in containerized clouds : a comprehensive analysis of open-source power measurement tools. *COMPUTER NETWORKS* 245, Article 110371 (2024). http://doi.org/10.1016/j.comnet.2024.110371

[6] Docker. 2022. Deploy to Swarm. https://docs.docker.com/get-started/swarm-deploy/

[7] Jordi Guitart. 2024. Practicable live container migrations in high performance computing clouds: Diskless, iterative, and connection-persistent. *Journal of Systems Architecture* 152 (2024), 103157. doi:10.1016/j.sysarc.2024.103157

[8] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*.

[9] Shengyi Huang and Santiago Ontañón. 2022. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. *The International FLAIRS Conference Proceedings* 35 (May 2022). doi:10.32473/flairs.v35i.130584

[10] Paulo Souza Junior, Daniele Miorandi, and Guillaume Pierre. 2022. Good Shepherds Care For Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes . In *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*. IEEE Computer Society, Los Alamitos, CA, USA, 26–33. doi:10.1109/ICFEC54809.2022.00011

[11] Kubernetes. 2022. Production-Grade Container Orchestration. https://kubernetes.io/

[12] Ning Liu, Zhe Li, Jielong Xu, Zhiyuan Xu, Sheng Lin, Qinru Qiu, Jian Tang, and Yanzhi Wang. 2017. A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 372–382. doi:10.1109/ICDCS.2017.123

[13] Peini Liu, Joan Oliveras Torra, Marc Palacín, Jordi Guitart, Josep Ll. Berral, and Ramon Nou. 2024. Data-Connector: An Agent-Based Framework for Autonomous ML-Based Smart Management in Cloud-Edge Continuum. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*. 1–6. doi:10.1109/ICNP61940.2024.10858515

[14] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (New York, NY, USA) *(ICML'16)*. JMLR.org, 1928–1937.

[15] Andres F. Ocampo and José Santos. 2024. Reinforcement Learning-Driven Service Placement in 6G Networks across the Compute Continuum. In *2024 20th International Conference on Network and Service Management (CNSM)*. 1–9. doi:10.23919/CNSM62983.2024.10814365

[16] Benoit Petit. 2023. *scaphandre*. https://github.com/hubblo-org/scaphandre

[17] Bjorn Pijnacker, Brian Setz, and Vasilios Andrikopoulos. 2025. Container-level Energy Observability in Kubernetes Clusters. arXiv:2504.10702 [cs.DC] https://arxiv.org/abs/2504.10702

[18] Leonardo Poggiani, Carlo Puliafito, Antonio Virdis, and Enzo Mingozzi. 2024. Live Migration of Multi-Container Kubernetes Pods in Multi-Cluster Serverless Edge Systems. In *Proceedings of the 1st Workshop on Serverless at the Edge* (Pisa, Italy) *(SEATED '24)*. Association for Computing Machinery, New York, NY, USA, 9–16. doi:10.1145/3660319.3660330

[19] Hongshuai Ren, Yang Wang, Chengzhong Xu, and Xi Chen. 2020. SMig-RL: An Evolutionary Migration Framework for Cloud Services Based on Deep Reinforcement Learning. *ACM Trans. Internet Technol.* 20, 4, Article 43 (Oct. 2020), 18 pages. doi:10.1145/3414840

[20] José Santos, Efstratios Reppas, Tim Wauters, Bruno Volckaert, and Filip De Turck. 2025. Gwydion: Efficient auto-scaling for complex containerized applications in Kubernetes through Reinforcement Learning. *Journal of Network and Computer Applications* 234 (2025), 104067. doi:10.1016/j.jnca.2024.104067

[21] José Santos, Mattia Zaccarini, Filippo Poltronieri, Mauro Tortonesi, Cesare Stefanelli, Nicola Di Cicco, and Filip De Turck. 2025. HephaestusForge: Optimal microservice deployment across the Compute Continuum via Reinforcement Learning. *Future Generation Computer Systems* 166 (2025), 107680. doi:10.1016/j.future.2024.107680

[22] Santos, José and Reppas, E. and Wauters, Tim and Volckaert, Bruno and De Turck, Filip. 2025. Can reinforcement learning be generalized for efficient auto-scaling in containerized clouds?. In *NOMS2025, the IEEE/IFIP Network Operations and Management Symposium* (Honolulu, USA). 7.

[23] Zhiqing Tang, Xiaojie Zhou, Fuming Zhang, Weijia Jia, and Wei Zhao. 2019. Migration Modeling and Learning Algorithms for Containers in Fog Computing. *IEEE Transactions on Services Computing* 12, 5 (2019), 712–725. doi:10.1109/TSC.2018.2827070

[24] Nassima Toumi, Miloud Bagaa, and Adlen Ksentini. 2023. Machine Learning for Service Migration: A Survey. *IEEE Communications Surveys Tutorials* 25, 3 (2023), 1991–2020. doi:10.1109/COMST.2023.3273121

[25] Indrani Vasireddy, Rajeev Wankar, and Raghavendra Rao Chillarige. 2024. Pod Migration with Optimized Containers Using Persistent Volumes in Kubernetes. In *Proceedings of World Conference on Information Systems for Business Management*, Andres Iglesias, Jungpil Shin, Bharat Patel, and Amit Joshi (Eds.). Springer Nature Singapore, Singapore, 27–36.

[26] Heng Zhang, Shaoyuan Huang, Mengwei Xu, Deke Guo, Xiaofei Wang, Victor C.M. Leung, and Wenyu Wang. 2023. How Far Have Edge Clouds Gone? A Spatial-Temporal Analysis of Edge Network Latency In the Wild. In *2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*. 1–10. doi:10.1109/IWQoS57198.2023.10188741

[27] Hansheng Zhang, Song Wu, Hao Fan, Zhuo Huang, Weibin Xue, Chen Yu, Shadi Ibrahim, and Hai Jin. 2025. KubeSPT: Stateful Pod Teleportation for Service Resilience With Live Migration. *IEEE Transactions on Services Computing* 18, 3 (2025), 1500–1514. doi:10.1109/TSC.2025.3564888