Journal of Network and Computer Applications 234 (2025) 104067

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

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

Research paper

Gwydion: Efficient auto-scaling for complex containerized applications in Kubernetes through Reinforcement Learning

José Santos [a,*], Efstratios Reppas [b], Tim Wauters [a], Bruno Volckaert [a], Filip De Turck [a]

[a] Ghent University - imec, IDLab, Department of Information Technology, Technologiepark - Zwijnaarde 126, Gent, 9052, Belgium
[b] National Technical University of Athens (NTUA), Greece

ARTICLE INFO

Keywords:
Auto-scaling
Containers
Cloud-native
Orchestration
Reinforcement Learning
Kubernetes

ABSTRACT

Containers have reshaped application deployment and life-cycle management in recent cloud platforms. The paradigm shift from large monolithic applications to complex graphs of loosely-coupled microservices aims to increase deployment flexibility and operational efficiency. However, efficient allocation and scaling of microservice applications is challenging due to their intricate inter-dependencies. Existing works do not consider microservice dependencies, which could lead to the application's performance degradation when service demand increases. As dependencies increase, communication between microservices becomes more complex and frequent, leading to slower response times and higher resource consumption, especially during high demand. In addition, performance issues in one microservice can also trigger a ripple effect across dependent services, exacerbating the performance degradation across the entire application. This paper studies the impact of microservice inter-dependencies in auto-scaling by proposing *Gwydion*, a novel framework that enables different auto-scaling goals through Reinforcement Learning (RL) algorithms. *Gwydion* has been developed based on the OpenAI Gym library and customized for the popular Kubernetes (K8s) platform to bridge the gap between RL and auto-scaling research by training RL algorithms on real cloud environments for two opposing reward strategies: cost-aware and latency-aware. *Gwydion* focuses on improving resource usage and reducing the application's response time by considering microservice inter-dependencies when scaling horizontally. Experiments with microservice benchmark applications, such as Redis Cluster (RC) and Online Boutique (OB), show that RL agents can reduce deployment costs and the application's response time compared to default scaling mechanisms, achieving up to 50% lower latency while avoiding performance degradation. For RC, cost-aware algorithms can reduce the number of deployed pods (2 to 4), resulting in slightly higher latency (300 μs to 6 ms) but lower resource consumption. For OB, all RL algorithms exhibit a notable response time improvement by considering all microservices in the observation space, enabling the sequential triggering of actions across different deployments. This leads to nearly 30% cost savings while maintaining consistently lower latency throughout the experiment. Gwydion aims to advance auto-scaling research in a rapidly evolving dynamic cloud environment.

1. Introduction

Microservice architectures have emerged as the dominant approach for application deployment in modern cloud platforms (Newman, 2021). This architectural approach involves breaking down the traditional monolithic application into loosely-coupled microservices, implemented and deployed independently. This paradigm shift offers enhanced deployment flexibility, scalability, service portability, and operational efficiency (Li et al., 2021). However, orchestrating modern applications in today's cloud platforms is challenging due to their intricate microservice inter-dependencies. The widespread adoption of containers necessitates efficient deployment and orchestration strategies for applications in popular cloud platforms, such as Amazon ECS (Amazon, 2024a), K8s (Burns et al., 2019), and Red Hat OpenShift (Hat, 2024). Moreover, the next generation of applications, including Extended Reality (XR), Industrial Internet of Things (IIoT), and autonomous vehicles (e.g., cars and Unmanned Aerial Vehicles (UAVs)) add further complexity and put even more pressure on current cloud infrastructures (Giordani et al., 2020; Santos et al., 2021). The deployment of these applications is hindered by the inability of modern infrastructures and protocols to meet their stringent requirements, such as high reliability, low latency, and high bandwidth.

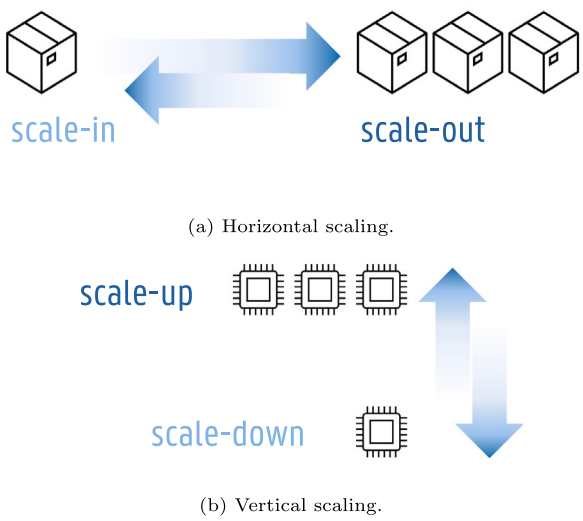
* Corresponding author.
E-mail address: josepedro.pereiradossantos@ugent.be (J. Santos).

https://doi.org/10.1016/j.jnca.2024.104067
Received 12 April 2024; Received in revised form 19 September 2024; Accepted 11 November 2024
Available online 26 November 2024

Container orchestration platforms typically support rapid adjustments to application deployment through *horizontal* and *vertical* scaling (illustrated in Fig. 1). Horizontal scaling involves increasing (scale-out) or decreasing (scale-in) the number of deployed instances (i.e., containers), and vertical scaling entails adjusting the resources allocated to each container instance (i.e., scale-up or scale-down). Over-provisioning wastes resources and increases costs, while under-provisioning degrades performance and violates Service Level Agreements (SLAs). Therefore, efficient auto-scaling (Qu et al., 2018) involves dynamically adding or removing resources to meet Quality of Service (QoS) requirements without human intervention. Designing efficient auto-scaling systems is challenging due to limited hardware resources, dynamic workloads, diverse service requirements, and complex infrastructures. Existing literature often focuses on either horizontal or vertical elasticity. Vertical scaling allows rapid responses to small workload variations, while horizontal scaling handles sudden workload peaks. However, most works mainly address resource utilization in the infrastructure (e.g., CPU and Memory), which is insufficient to satisfy the demanding requirements of microservice applications, especially concerning latency and bandwidth. As inter-dependencies increase, microservice communication becomes more complex and frequent, resulting in slower response times and increased resource consumption, particularly under high demand. Additionally, the strong interdependence between microservices means that performance issues in one service can trigger a cascading effect (Soldani et al., 2021), exacerbating degradation across the entire application. Current approaches, such as (AWS, 2024; Kubernetes, 2024a,b; Rattihalli et al., 2019; Srirama et al., 2020), typically scale containers for each microservice independently without considering their inter-dependencies.

This paper focuses on understanding the impact of microservice inter-dependencies on auto-scaling mechanisms. The goal is to identify optimal states for each microservice based on the current demand, considering its performance impact on the overall application pipeline. To this end, *Gwydion*,[1] a RL-based auto-scaling framework has been developed. The framework is inspired on the OpenAI Gym library (Brockman et al., 2016) to train RL agents with different auto-scaling objectives on operational cloud environments established with the most popular container orchestration platform, K8s (Burns et al., 2022). K8s automates various processes throughout the application lifecycle, including deployment and scaling. Traditional approaches mainly focus on threshold-based or Machine Learning (ML)-based methods (e.g., (Kubernetes, 2024a,b; Rossi et al., 2019; Lee et al., 2020; Rzadca et al., 2020)), focusing on resource efficiency without considering the application's response time or latency. This work addresses mainly horizontal scaling, as vertical scaling can trigger potentially costly operations. Adjusting container resources could lead to performance degradation or Out of Memory (OOM) errors, as containers may no longer fit on their machines. The main contributions of this paper are:

- ***Gwydion* framework and improved RL design**: Building upon our previous work (Santos et al., 2023b), the *Gwydion* framework significantly extends the *gym-hpa* framework[2] by adding predictive capabilities and refining the RL-based approach for horizontal scaling of microservices in K8s clusters. Section 5 presents the refined RL design, including observation state, action space, and reward functions. This approach addresses microservice inter-dependencies and the application's response time, both often neglected in most works, which are crucial for achieving efficient scaling decisions. *Gwydion* has also been open-sourced,[3] enabling researchers to leverage the framework to evaluate their auto-scaling concepts.
- **Integration of Resource Predictions in RL**: *Gwydion* adopts statistical and ML-based algorithms for predicting CPU and Memory usage of microservice applications. A microservice application named as *Gwydion-estimator* has been developed to provide real-time resource predictions, allowing *Gwydion* to label microservices accordingly. Results show that enabling predictive capabilities in *Gwydion* leads to faster learning at a slightly higher performance depending on the selected objective (Section 8.2).
- **Performance Evaluation with Microservice Benchmarks**: *Gwydion* has been validated on real-world microservice benchmark applications: a database application named as RC (Redis, 2024) (Fig. 2(a)), and a multi-tier web application denoted as OB (Boutique, 2024) (Fig. 2(b)). Experiments conducted in a K8s cluster demonstrate that the presented RL approach can reduce the latency up to 50% while avoiding performance degradation, more noticeably for the OB application. In addition, *Gwydion* can significantly reduce deployment costs for both applications, as shown in Section 8.



(a) Horizontal scaling.

(b) Vertical scaling.

Fig. 1. Illustration of horizontal and vertical scaling.

The remainder of the paper is organized as follows: the state-of-the-art on auto-scaling is discussed in the next section. Section 3 discusses application deployment and auto-scaling operations in K8s, describing its terminology. Section 4 details the *Gwydion* framework and Section 5 presents the RL-based auto-scaling approach. Section 6 details the prediction algorithms of *Gwydion*, highlighting the importance of accurate resource estimations. Then, Section 7 describes the evaluation setup, followed by the results in Section 8. Lastly, Section 9 focuses on open challenges and future directions, and Section 10 concludes this paper.

2. Related work

Recent surveys (Qu et al., 2018; Singh et al., 2019) provide comprehensive insights into auto-scaling features applied to cloud-based systems. Both surveys categorize auto-scaling approaches based on various criteria. This section explores the literature through five key dimensions: threshold-based, queuing model-based, time series analysis, control theory-based, and ML-based methods.

Threshold-based techniques (AWS, 2024; Kubernetes, 2024a,b; Rattihalli et al., 2019; Srirama et al., 2020) are widely adopted by the industry. Popular orchestration platforms (e.g., K8s, Amazon ECS) rely on best-effort threshold-based scaling policies based on cluster-level metrics such as CPU usage and the average number of requests. Most

[1] *Gwydion* is a magician, hero and trickster of Welsh mythology, which can be metaphorically linked to the complexity and creativity involved in auto-scaling research. Efficient auto-scaling algorithms need intricate design and optimization, related to Gwydion, the master of magic.

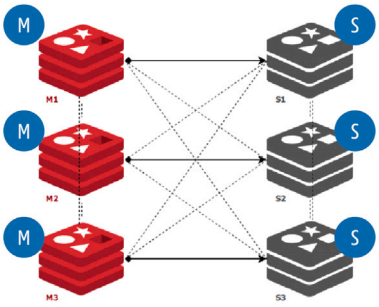
[2] https://github.com/jpedro1992/gym-hpa

[3] https://github.com/jpedro1992/gwydion

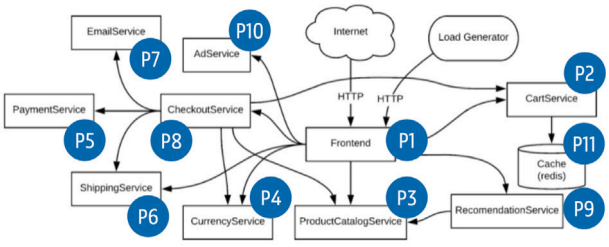
Table 1
Comparison of existing works related to auto-scaling.

Existing work	Dimension	Type	Policy	VT	Metrics	Dep.	Eval.
Amazon EC2 (AWS, 2024)	T	H	R	VMs + C	e.g., CPU + RAM	×	A
KHPA (Kubernetes, 2024a)	T	H	R	C	e.g., CPU, RAM	×	K
KVPA (Kubernetes, 2024b)	T	V	R	C	e.g., CPU, RAM	×	K
Rattihalli et al. (2019)	T	V	R	C	CPU + RAM	×	A + K
Srirama et al. (2020)	T	H	R	VMs + C	e.g., CPU + RAM	×	S
Gergin et al. (2014)	Q	H	R	VMs	RT	×	A
Danilo et al. (2021)	Q	H	R	VMs	CPU + RT	×	S + T
Calheiros et al. (2015)	TS	H	P	VMs	e.g, CPU, RT	×	S
Messias et al. (2015)	TS	H	R + P	VMs	RT	×	S
Kumar and Singh (2018)	TS	H	R + P	VMs	R	×	S
Farokhi et al. (2016)	CT	V	R	VMs	e.g., RAM, RT	×	T
Nouri et al. (2019)	CT + ML	H	R	VMs	e.g., CPU, RT	×	T
Toosi et al. (2019)	CT	H + V	R	VMs	e.g., CPU, TL	✓	S
Toka et al. (2021)	ML + TS	H	R + P	C	CPU + R	×	K
Rossi et al. (2019)	ML	H + V	R + P	C	e.g., CPU, RAM	×	S + D
Lee et al. (2020)	ML	H	P	VMs	T + RT	✓	O
Rzadca et al. (2020)	ML + TS	H + V	P	C	e.g., CPU, RAM	×	T
Santos et al. (2023b)	T + ML	H	R + P	C	e.g., RT, CPU, RAM	✓	S + K
Gwydion	**T + TS + ML**	**H**	**R + P**	**C**	**RT + CPU + RAM**	✓	**S + K**

Dimension: T = Threshold, Q = Queuing model, TS = Time Series analysis, CT = Control theory, ML = ML-based.
Type: H = Horizontal, V = Vertical.
Policy: R = Reactive, P = Proactive.
Virtualization (VT): VMs = Virtual Machines, C = Containers.
Metrics: CPU = CPU usage, RAM = Memory usage, T = Throughput, RT = Response time, R = Number of requests.
Dependencies (Dep.): ✓ = addressed, × = not considered.
Evaluation (Eval.): A = Amazon AWS, K = Kubernetes, D = Docker, O = Openstack, S = Simulation, T = Testbed.



(a) Redis Cluster (RC) application.



(b) Online Boutique (OB) application.

Fig. 2. Illustration of microservice dependencies (Santos et al., 2023b).

cloud providers offer reactive threshold-based scaling methods, such as Amazon EC2 (AWS, 2024) and Kubernetes Horizontal Pod Autoscaler (KHPA). Amazon EC2 is particularly well suited for applications with predictable traffic patterns (e.g., daily or weekly) as it allows tailored rules to handle these patterns. KHPA adjusts the number of pods based on specific metrics (e.g., CPU usage) by maintaining an average of the desired metric and scaling pods accordingly. However, a notable drawback of these methods is the manual tuning required to determine appropriate thresholds and scaling actions, which can significantly impact application performance if poorly chosen (Qu et al., 2018). On the other hand, vertical scaling techniques (Kubernetes, 2024b; Rattihalli et al., 2019) involve resizing containers dynamically during runtime. Kubernetes Vertical Pod Autoscaler (KVPA) sets container resource limitations based on statistics gathered over a moving window (e.g., the 99th percentile of memory usage over 24 h). *Gwydion* focuses on horizontal scaling, as resizing containers may lead to OOM errors or compromise QoS during sudden usage spikes. We argue that vertical scaling is a risky procedure that needs further research to ensure that dynamically adapting container resources at runtime does not compromise performance.

Queuing model-based methods are widely used for analyzing Internet applications, particularly for estimating performance metrics and request waiting times. These techniques can understand the behavior and performance of the system under various conditions. Recent works (Gergin et al., 2014; Danilo et al., 2021) have applied queuing theory to auto-scaling strategies. The primary objective of these methods is to optimize deployment costs while ensuring that the application's Service Level Objectives (SLOs) are maintained. However, it is important to note that queuing models are generally designed for analyzing stationary systems, assuming that demand remains constant over time. In the current cloud landscape, numerous real-world applications experience dynamic workloads with fluctuating demands. The models rely on known parameters (e.g., the arrival rate of service requests), making them effective in predicting system behavior under stable conditions. When facing dynamic workloads, the known parameters may no longer accurately represent the system, requiring model and metric recalibration to match the current dynamics. Adapting queuing model-based methods to handle dynamic microservice applications remains an enormous challenge, enhancing their suitability for real-world environments.

Time series analysis typically comprises a two-step process: forecasting workload patterns and triggering scaling actions based on the predicted workload. However, a majority of existing methods (Calheiros et al., 2015; Messias et al., 2015; Kumar and Singh, 2018) rely on predefined thresholds to trigger actions when the predicted metric surpasses a certain threshold. Workload prediction models are commonly applied to produce efficient scaling actions, aiming for optimal resource usage while minimizing adverse impacts on QoS (Calheiros et al., 2015). Nonetheless, the applicability of these methods across diverse workload types remains challenging. Predicting incoming requests (Messias et al., 2015) or subsequent resource consumption can be time-consuming, potentially limiting real-time responsiveness.

Furthermore, these approaches often focus on individual microservices, neglecting the complex dependencies within a microservice-based application. In contrast, *Gwydion* advocates for determining appropriate scaling actions based on the real-time status of multiple microservices, offering a more comprehensive and adaptive solution to handle the intricate dynamics of microservice applications.

Control theory-based methods (Farokhi et al., 2016; Nouri et al., 2019; Toosi et al., 2019) typically consist of two main phases (i.e., analysis and planning) coming from the Monitoring, Analysis, Planning and Execution (MAPE) loop (Arcaini et al., 2020). These phases work in tandem to understand system behavior and make informed adjustments. The essence lies in modifying the system's behavior to align with specified output and reference values, aiming to bridge discrepancies through responsive feedback mechanisms. In auto-scaling, the desired SLA serves as the reference value, while performance metrics such as CPU usage represent the output. Experiments show that these approaches significantly reduce deployment costs by enhancing overall memory utilization (Farokhi et al., 2016) and by mitigating SLA violations (Toosi et al., 2019). However, these techniques depend highly on the controller design and the target application. Their main drawback occurs when dealing with dynamic and unpredictable workloads. Existing works tackle this challenge by combining control-theory methods with ML-based techniques or leverage time series analysis to predict future demands within the controller and adapt resources accordingly. The main advantage of these hybrid approaches is their potential to significantly reduce execution time and enhance the overall efficiency of the scaling process.

ML-based techniques (Rossi et al., 2019; Lee et al., 2020; Rzadca et al., 2020; Santos et al., 2023b; Toka et al., 2021) are gaining substantial traction in modern auto-scaling strategies. The primary objective of these techniques is to build a model for accurately estimating resource needs under specific workloads. These approaches are robust to dynamic demands since the algorithm adjusts the model parameters if any notable event occurs (i.e., online learning). The model also can be trained offline, but it would often require significant human intervention, losing the main benefit of these algorithms. Google Autopilot, introduced in Rzadca et al. (2020), automatically configures resources, adjusting the number of concurrent tasks in a job (i.e., horizontal scaling) and the CPU/RAM limits for individual tasks (i.e., vertical scaling). Autopilot aims to reduce the difference between resource limits and actual resource usage, thereby reducing the risk of OOM errors and performance degradation due to CPU throttling. Autopilot applies ML algorithms to analyze historical data and discern patterns from past task executions. Results show that Autopilot significantly reduces resource utilization and minimizes OOM errors. The main drawback of ML-based approaches is the high execution time to converge to a stable model and thus causes auto-scaling to perform suboptimally during the learning phase. Recent efforts focus on optimizing the execution time while ensuring the accuracy and stability of the resulting ML models. The potential of ML-based auto-scaling strategies to revolutionize application performance makes this active research field highly promising for future cloud computing environments.

Table 1 compares all methods introduced in this section, classifying them according to their main characteristics. However, quantitatively evaluating these methods poses a challenge due to their tailored design for specific systems or virtualization technologies. To the best of our knowledge, no standard testing framework for evaluating auto-scaling features exists. In previous work, we have proposed scheduling extensions for the K8s platform (Santos et al., 2019, 2023a) that address complex microservice-based applications. This paper extends those efforts by focusing on the other important aspect of the life cycle management of containerized applications: auto-scaling. Moreover, this work builds further upon our previous work (Santos et al., 2023b), in which the *gym-hpa* framework has been presented. *Gwydion* extends *gym-hpa* by integrating prediction algorithms and refining the RL-based approach for horizontal scaling of microservices in K8s clusters

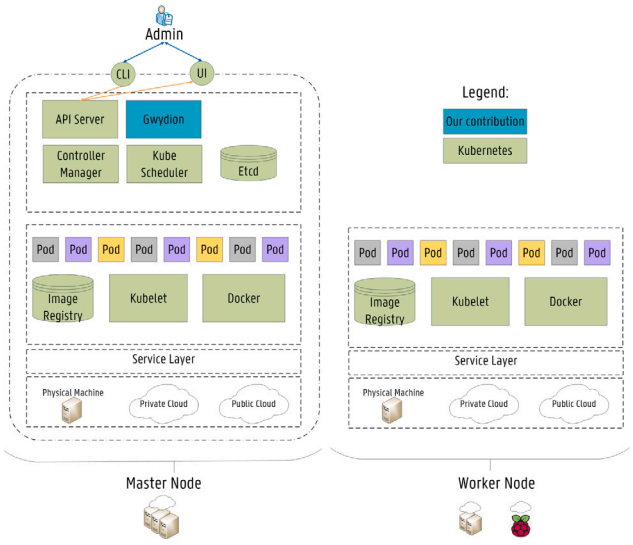


Fig. 3. The Kubernetes architectural model (Santos et al., 2019).

based on lessons learned from previous experiments (Santos et al., 2023b). The work differs from the current literature by addressing microservice inter-dependencies and combining prediction capabilities with RL algorithms for a more efficient auto-scaling solution. *Gwydion* aims to serve as an open-source initiative towards a more comprehensive understanding and effective utilization of auto-scaling mechanisms in the K8s platform, aligned with the intricate dynamics of modern microservice-based applications.

3. Application deployment and auto-scaling in the Kubernetes (K8s) platform

The proliferation of microservice patterns and the widespread adoption of containers have lead to numerous orchestration platforms, both from the industry and the open-source community. Among these platforms, K8s has emerged as the most prominent, offering a rich set of software components to automate the life cycle management of containerized applications across distributed cluster nodes. K8s operates on the well-established master–slave model, where at least one master node oversees containers distributed across multiple worker nodes, also known as slaves (Fig. 3). Master nodes typically possess higher computational resources to host various critical software components such as the *Application Programming Interface (API) server*, *Kubelet*, and *Controller Manager*, responsible for orchestrating the complete life cycle workflow of containerized applications. One of the main benefits of K8s is that its API is standardized across vendors, with applications capable of being deployed on private data center resources or managed public vendor options such as Amazon EKS (Amazon, 2024b), and Google Kubernetes Engine (GKE) (Google, 2024).

Microservices within K8s are often tightly coupled into a group of containers known as a *pod*. A *pod* represents the smallest functional unit in K8s and includes a collection of containers and volumes (i.e., storage) executing within the same runtime environment (Burns et al., 2019). K8s establishes connections between identical pods via a *Deployment* (Kubernetes, 2022a) entity, but it does not natively support the aggregation of distinct pods into a particular application. Thus, developers need to configure individual KHPA components for each microservice (i.e., *Deployment*) of their application for proper horizontal scaling across their entire application. This results in KHPA handling horizontal scaling for each *Deployment* without any knowledge about its microservice inter-dependencies. Current auto-scaling strategies in K8s do not consider the holistic view of the application, which could lead

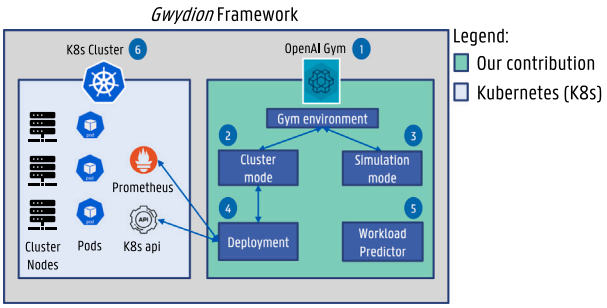


Fig. 4. Illustration of the *Gwydion* framework.

to suboptimal scaling decisions. The following section describes the *Gwydion* framework, a novel auto-scaling approach that acknowledges microservice dependencies by aggregating multiple K8s deployments into an application entity. Therefore, *Gwydion* aims to study the complex relationships and interactions between microservices within a microservice-based application, offering comprehensive insights to the research community.

4. The *Gwydion* framework

4.1. System overview

Fig. 4 provides an overview of the *Gwydion* framework, illustrating its key software components and their interconnections. The foundation of this framework lies in the OpenAI Gym module ①, a vital building block that offers a wide range of gym environments for enabling RL training. *Gwydion* supports two evaluation modes during startup: cluster ②and simulation ③.

The cluster mode involves RL training within an actual K8s cluster environment, enabled by the deployment component ④. This component interacts with a K8s cluster ⑥through the K8s API to retrieve essential information about the target application. Moreover, it also leverages the Prometheus API (Turnbull, 2018), a widely-recognized monitoring platform typically integrated natively within K8s, to gather real-time usage metrics of the application by collecting samples for all its microservices. On the other hand, the simulation mode represents a near-real discrete-event experiment aimed at replicating the behavior of multiple service requests for a specific application deployed on the K8s platform. This mode accelerates the learning process by training RL agents based on data collected during the cluster mode execution. Therefore, the simulation mode can be executed without a K8s cluster, making the RL training significantly faster.

During the simulation mode, the application's resource usage metrics, such as CPU and memory usage, and the number of deployed pods are dynamically updated based on scaling actions and the current service demand. Section 4.3 provides further insights into how datasets have been created based on microservice-based applications to support the simulation mode and ensure the simulation closely emulates real-world experiments. Another vital component is the Workload Predictor ⑤, which deploys the *Gwydion-estimator* pod for each microservice. This estimator utilizes Deep Learning (DL) algorithms to forecast CPU and memory usage, effectively labeling each microservice with the expected resource usage. Further explanations are given below on how the Deployment component interacts with K8s, detailing the underlying mechanisms that enable *Gwydion* to effectively harness and integrate real-world application data for robust RL training.

Table 2
Deployment status in the *gym-hpa* framework.

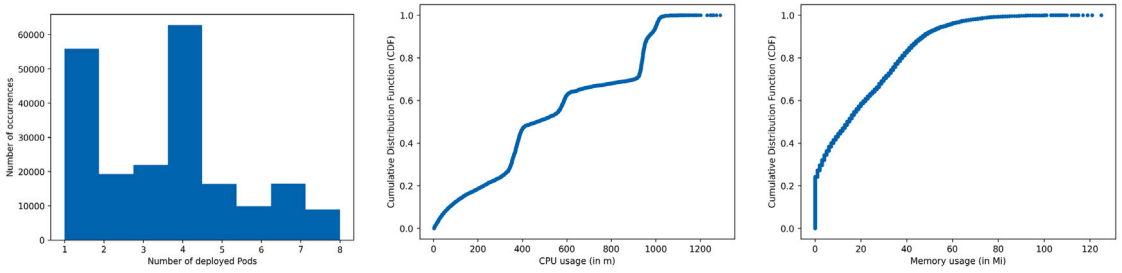
Symbol	Description
A	The application a. Each application consists of a set of different deployments $d \in D_a$.
D_a	The set of deployments belonging to the application a.
C_d	The set of container names $c \in C$ for deployment d.
P_d	The set of all pods belonging to the deployment $d \in D$.
$\alpha_{d,max}$	The maximum replication allowed for deployment d.
$\alpha_{d,min}$	The minimum replication allowed for deployment d.
$\gamma_{d,[r]}$	The request vector of deployment d. r denotes resources as CPU (in m) and memory (in Mi). m stands for millicore and Mi stands for mebibyte.
$\Gamma_{d,[r]}$	The limit vector of deployment d. r denotes resources as CPU (in m) and memory (in Mi).
$\rho_{d,[r]}$	The total usage vector of deployment d. r denotes resources as CPU (in m) and memory (in Mi).
R_d	The current number of deployed pods for deployment d.
T	The threshold for resource usage. Default: 0.75.
Ω_c	The CPU weight for replica calculation. Default: 0.7.
Ω_m	The memory weight for replica calculation. Default: 0.3.
$\lambda_{[r]}$	The target resource usage vector. r denotes resources as CPU (in m) and memory (in Mi).
ω_d	The desired number of replicas for deployment d.
τ_a	The latency threshold for application a (in ms).
Ψ_a	The current application latency (in ms).

4.2. Integrating Gwydion with Kubernetes (K8s)

Table 2 presents a comprehensive overview of the information available within the Deployment component, derived from a K8s deployment. *Gwydion* allows developers to specify the microservices (D_a) that constitute an application (a). Input information (e.g., C_d, P_d) is retrieved from the K8s API, while its real-time status (e.g., R_d, $\rho_{d,[r]}$) is retrieved from the Prometheus API. Moreover, resource requests ($\gamma_{d,[r]}$) denote the minimum amount of resources (e.g., CPU, memory) required by all containers within a pod, while limitations ($\Gamma_{d,[r]}$) represent the maximum allocation of resources for these containers (Kubernetes, 2022b). Developers are encouraged to specify these resource requests and limits (R/L) in their deployments, enabling K8s to optimize scheduling and auto-scaling for the respective pods. It is noteworthy that container abstraction provides less isolation than Virtual Machines (VMs), and when multiple containers run on the same cluster node, the sharing of physical resources might result in performance degradation known as resource contention (Medel et al., 2018). Within the Deployment component, the desired number of replicas (R_d) is calculated based on the specified resource requests. The existing KHPA scales the number of pods in a deployment based on the resource usage of a particular metric, within the specified minimum and maximum replication thresholds (α_{max} and α_{min}). By default, KHPA focuses on CPU usage and determines the desired replica count according to (1). We argue that this formula must be adapted to consider several resource types. In the *Gwydion* framework, the number of desired replicas is calculated as in (4), integrating target usages for each resource (2) and their respective influence on the precise replica count (3). Typically, a target resource usage of 75% is set as the default, as aiming for optimal usage (i.e., 100% resource utilization) might lead to performance degradation in the event of sudden demand spikes or if containers request additional computing resources. Regarding the application's latency (Ψ_a), researchers have the flexibility to specify the particular measurement or metric to consider. Section 7.1 describes the two evaluated applications, alongside the corresponding measurements adopted to represent the application's latency.

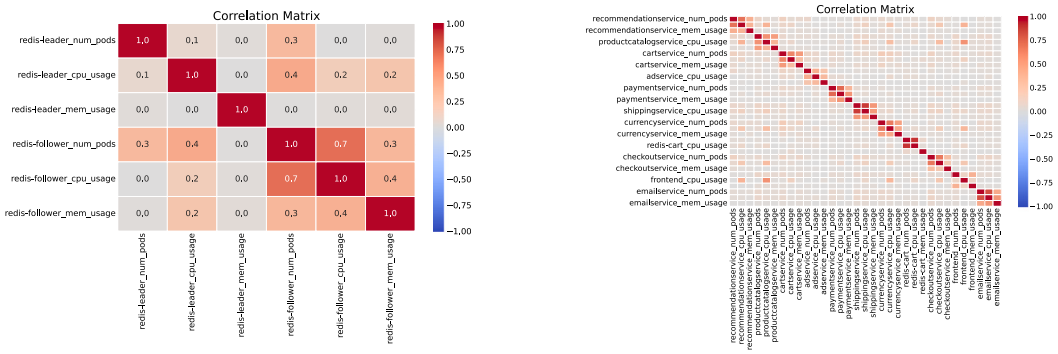
$$\underbrace{\omega_d = \lceil R_d * \left(\frac{\rho_{d,cpu}}{\lambda_{cpu}}\right) \rceil}_{\text{desired replicas for default KHPA}} \tag{1}$$

$$\underbrace{\lambda_{[r]} = R_d \times \left(T \times \gamma_{d,[r]}\right)}_{\text{The target number for resource } r} \tag{2}$$



(a) Number of Deployed pods. (b) CDF (CPU Usage). (c) CDF (Memory Usage).

Fig. 5. Analysis of the *leader* microservice of the Redis Cluster (RC) application (Santos et al., 2023b).



(a) Redis Cluster (RC) application. (b) Online Boutique (OB) application.

Fig. 6. The correlation matrix for both evaluated applications. A strong correlation typically exists between the CPU usage of the microservice and the corresponding number of pods deployed in the cluster.

$$\underbrace{\omega_{[r]} = \lceil R_d \times \frac{\rho_{d,[r]}}{\lambda_{[r]}} \rceil}_{\text{The desired number of replicas based on resource } r} \tag{3}$$

$$\underbrace{\omega_d = \lceil \underbrace{\Omega_c}_{\text{CPU weight}} \times \underbrace{\omega_c}_{\text{desired CPU}} + \underbrace{\Omega_m}_{\text{Mem. weight}} \times \underbrace{\omega_m}_{\text{desired Mem.}} \rceil}_{\text{desired replicas for } \textit{Gwydion}} \tag{4}$$

4.3. Dataset creation for the simulation mode

Datasets can be collected for applications from real K8s deployments by generating different requests to trigger several scaling actions. These datasets are saved in Comma Separated Value (CSV) files to help build a tailored simulation where each observation corresponds to the actual resource usage of these applications in a K8s cluster. Consequently, when an RL agent selects a particular action, an appropriate observation is retrieved from the dataset. This approach transforms the simulation mode into a near-real experiment, closely mirroring the behavior and dynamics of the actual applications running in K8s clusters. As an illustrative example, Fig. 5 shows the number of deployed pods and the corresponding Cumulative Distribution Functions (CDFs) regarding CPU and Memory usage based on the master deployment of the RC application. Fig. 6 illustrates the correlation matrix of both evaluated applications (further details are given in Section 7). It represents the relationship between different variables in a dataset with a correlation coefficient, in which 1.0 is considered a strong relationship, and −1.0 a poor relationship (i.e., negatively correlated). As shown, the CPU usage of the microservice is typically strongly related to the number of pods deployed in the cluster. Also, it should be noted that the number of pods in the *follower* is strongly related to the CPU usage of the *leader*. A few microservices strongly impact the performance of dependent microservices, as shown by the strong correlation between the *frontend* CPU usage and the *product-catalog* CPU usage.

5. Towards efficient auto-scaling in Kubernetes (K8s)

5.1. Reinforcement Learning (RL)-based auto-scaling

RL has recently become an important research field (Hessel et al., 2018), often applied to solve sequential decision-making problems. In RL, agents learn to select actions directly from experience by interacting with an environment. At first, the agent knows nothing about the task and learns by receiving a reward for each action. The reward is related to the new observation, which describes the environment state after applying the action selected by the agent. For example, in auto-scaling, the reward is positive if the agent's action increases application performance (e.g., efficient resource usage, low response time). In contrast, the agent receives a negative reward if the performance degrades. The agent learns to perform the task by repeated interactions with the environment and determining the inherent synergies between states, actions, and subsequent rewards. The goal of RL is to teach an agent to select actions that maximize application performance and minimize deployment costs. RL is well-suited for auto-scaling problems because agents can continuously receive feedback from the environment and adjust their action selection to achieve long-term objectives in complex situations, such as microservice auto-scaling. The following subsections describe the RL approach applied in *Gwydion* for solving horizontal auto-scaling of complex microservice applications in K8s.

5.2. Action space

The action space corresponds to all actions that the agent can perform in the environment. Table 3 shows the action space structure of the *Gwydion* framework based on an application with two microservices (*D1* and *D2*). The action space is *MultiDiscrete* (openAIGym, 2024), where a list of possible actions per discrete set exists. However, the agent can only select one action per each discrete set per step. The first discrete set corresponds to microservice selection, and the second

Table 3
The action space structure applied in *Gwydion*.

Discrete set	Action	Description
Microservice	*D1*	Action triggered on Deployment 1.
	D2	Action triggered on Deployment 2.
Scaling	*DoNothing*	The agent does nothing.
	Start-1	Deploy one replica.
	Start-2	Deploy two replicas.
	Start-3	Deploy three replicas.
	Stop-1	Stop one instance.
	Stop-2	Stop two instances.
	Stop-3	Stop three instances.

Table 4
The *Gwydion's* Observation Space Structure.

Metric	Description
pods	The number of deployed pods.
cpu	The total aggregated CPU (in m) of the pods.
mem	The total aggregated memory (in Mi) of the pods.
counter	The number of *DoNothing* actions selected sequentially.

one to scaling actions. The size of the action space depends on the total number of microservices in the application and the correspondent maximum and minimum replication factor ($\alpha_{d,max}$ and $\alpha_{d,min}$). For example, if $\alpha_{d,max} = 4$ and $\alpha_{d,min} = 1$, the maximum number of additions or terminations that the agent can select for each microservice is three (i.e., second discrete set). Thus, agents can decide to:

- Keep the deployment running as is (*DoNothing*)
- Deploying additional pods (*Start*)
- Terminate a certain number of pods (*Stop*)

For instance, the agent can choose the following action $[Microservice : D1; Scaling : DoNothing]$, meaning that the number of pods for microservice *D1* will remain unchanged for that given step.

5.3. Observation space

Table 4 shows an example of the observation space for the auto-scaling problem in the *Gwydion* framework, describing the environment at a given step. The updated version of the observation space in *Gwydion* consists of three metrics per microservice deployment, namely the current number of deployed pods (*pods*), and the current resource utilization (*cpu* and *mem*). This information is retrieved from the Deployment component in case the cluster mode is enabled or from the simulation. The last considered metric is a none counter (*counter*) that represents the number of consecutive *DoNothing* actions selected by the agent. In previous experiments with *gym-hpa* (Santos et al., 2023b), we observed that the agents occasionally converged to suboptimal states, preferring to choose the *DoNothing* action instead of trying to find the optimal deployment scheme for each microservice, leading to suboptimal performance. As a resolution, the none counter metric is added to the observation space to encourage the agent not to overselect *DoNothing* actions, especially if the current deployment scheme is not optimal based on the current demand. All these metrics help the agent select an adequate action at a given moment from the action space.

Another improvement in *Gwydion* is the normalization of the observation space (*min–max* scaling is applied), a common practice in RL. Normalizing the observation space can help the RL agents to converge faster and more reliably. When observations have a consistent scale, it can be easier for neural networks to learn optimal policies. Also, it stabilizes the training for large input values, preventing the explosion of gradients. It should be noteworthy that the application latency is not included in the observation space. The aim is that the agent learns that the current distribution of microservices can yield lower response times, depending on the current resource usage of all microservices.

5.4. Reward function

The purpose of the reward function is to teach the agent to maximize its accumulated reward by selecting appropriate actions based on the environment's observation. Two reward functions have been designed, each with a different objective: cost-aware and latency-aware.

Cost-aware reward (*C*) (Eq. (5)) The cost function aims to lead the agent to allocate the correct number of replicas for each microservice deployment, focused on reducing deployment costs by increasing resource usage. It rewards the agent with a positive reward of 1 when the number of replicas deployed aligns with the desired configuration, as indicated by Eq. (4). Otherwise, the agent receives no positive feedback (i.e., reward of 0). In addition, if the agent attempts to deploy or terminate pod instances that would violate the maximum or minimum replication factor, it receives a penalty of −1. This penalty helps the agent learn the valid range of actions for pod deployment. The values of +1 and −1 have been empirically selected through trial-and-error. During our experiments, this balance provided effective learning for the agent. The symmetric nature of the rewards helped stabilize the learning process without overly penalizing the agent for replication factor violations.

$$C(d) = \begin{cases} 1.0 & \text{if } R_d = \omega_d \\ -counter & \text{if } R_d \neq \omega_d \wedge counter > 2 \\ 0 & \text{Otherwise} \end{cases} \tag{5}$$

Latency-aware reward (*L*) (Eq. (6)) The latency function guides the agent to find suitable allocation schemes that reduce the overall application latency (Ψ_a). The agent receives a penalty based on latency, with the goal of achieving the lowest possible reward (ideally zero) if latency remains below a specified threshold (τ_a). If latency exceeds this threshold, the agent is penalized more severely by receiving a negative reward of $-\tau_a$. Both reward functions follow a linear pattern since these were found to be more effective than exponential functions in our experiments, i.e. the agent learned to perform more adequate actions.

$$L(a) = \begin{cases} -\Psi_a & \text{if } \Psi_a \leq \tau_a \\ -counter \times \tau_a & \text{if } \Psi_a > \tau_a \wedge counter > 2 \\ -\tau_a & \text{Otherwise} \end{cases} \tag{6}$$

An additional penalty is introduced in *Gwydion* in comparison with the previous *gym-hpa*, in which the agent is significantly penalized if it prefers to keep the deployment scheme as is when the current demand indicates at least one addition or termination of a microservice instance. It occurs if the none counter is higher than two, leaving some room for the agent to explore the action space.

Cloud administrators face the ongoing challenge of balancing resource efficiency while minimizing latency in their cloud infrastructure. Their ultimate decisions are made based on user needs and organizational objectives. While prioritizing costs can help minimizing deployment expenses, particularly for budget-constrained companies, favoring low latency is crucial for real-time applications where responsiveness is key. This paper evaluates the trade-off of both strategies while showcasing their benefits and disadvantages.

5.5. Agents

The agents have been implemented based on the stable baselines 3 library (Raffin et al., 2021), a set of reliable RL algorithm implementations written in Python. Three agents that support *MultiDiscrete* action spaces have been evaluated in *Gwydion*:

- **Advantage Actor Critic (A2C)** (Mnih et al., 2016): A synchronous, deterministic algorithm that combines policy and value-based algorithms.
- **Proximal Policy Optimization (PPO)** (Engstrom et al., 2019): a policy gradient method for RL vastly used today for different scenarios (e.g., robot control and video games)

- **Recurrent Proximal Policy Optimization (RPPO)** (Ratcliffe et al., 2019): A variant of PPO that adds support for recurrent policies, such as Long Short-Term Memory (LSTM).

Policy-based agents directly learn a policy mapping input states to output actions, meaning agents choose the action in a given state based on its previous experience (i.e., actors). Value-based algorithms learn to select actions based on the predicted value of the input state or action (i.e., critic), meaning agents choose actions that lead to states with higher expected rewards. Also, recurrent policies can be beneficial for tasks where the agent's actions depend on its past observations. For example, in the microservice auto-scaling problem, the agent's next action may depend on the current and previous states of the microservices. Recurrent policies can learn these dependencies and make better decisions as a result.

6. Towards resource usage prediction

This section presents the inclusion of algorithms for future resource usage prediction added to the *Gwydion* framework. Section 6.1 details the prediction algorithms applied in the *Gwydion* framework, and Section 6.2 presents the *deployment-estimator* developed for K8s deployments.

6.1. Gwydion's prediction algorithms

To achieve a more efficient resource usage and proactive strategy in *Gwydion*, statistical and ML-based algorithms have been used to forecast future resource consumption of microservice applications. Recent works have shown the potential of several algorithms in estimating future resource consumption (Janardhanan and Barrett, 2017; Rao et al., 2019). *Gwydion* supports the following algorithms:

- **Naive** serves as a benchmark in *Gwydion* forecasting. This method involves predicting future values equal to the most recent observation. Despite its simplicity, it often proves surprisingly effective, especially in environments characterized by considerable randomness, frequently outperforming more complex and sophisticated forecasting methods.
- **Simple Exponential Smoothening (SES)** (Ostertagova and Ostertag, 2012) is a widely used time series forecasting method that provides an efficient way to make short-term predictions based on historical data. This method focuses on capturing the underlying trend and seasonality in time series data by assigning exponentially decreasing weights to past observations. It essentially gives more weight to recent data points while progressively reducing the influence of older data samples. SES is mainly applied for data with no significant trends or seasonality.
- **LSTM** (Yu et al., 2019) is a type of neural network designed for forecasting data sequences. Their main advantage is the ability to capture long-range dependencies and handle vanishing gradient problems that traditional neural networks struggle with. LSTM maintains a memory cell that can store and retrieve information over extended sequences, making them well-suited for tasks such as time series prediction, speech recognition, and language modeling.
- **Auto-Regressive Integrated Moving Average (ARIMA)** (Shumway et al., 2017) models are a widely popular time series forecasting method that captures temporal patterns and trends in data. These models predict based on stationary timeseries using weights on previous values. The *Auto-Regressive* component models the relationship between the current value, and previous values in the time series. The *Integrated* part represents differencing to stabilize and make the time series stationary, while the *Moving Average* component models the dependency on past forecast errors. ARIMA models are versatile and can handle both short-term and long-term time series forecasting by adjusting the order of these components.

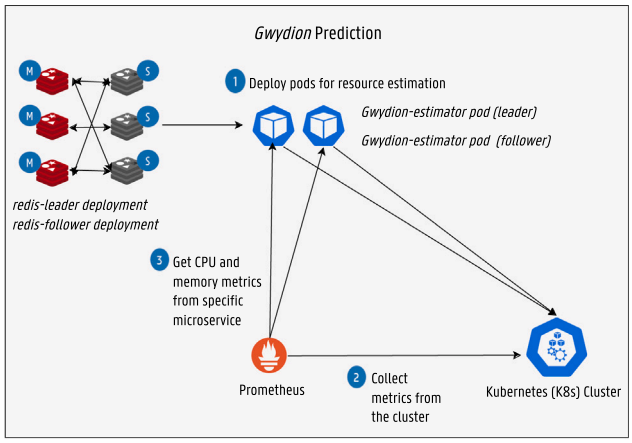


Fig. 7. Illustration of the *Gwydion* prediction capabilities.

```
josperei@master:~/auto-scaling/manifests/redis$ kubectl describe deployment redis-leader
Name:                   redis-leader
Namespace:              default
CreationTimestamp:      Mon, 23 Oct 2023 09:59:39 +0200
Labels:                 app=redis-leader
                        role=leader
                        tier=backend
Annotations:            deployment.kubernetes.io/revision: 1
                        forecast-arima-cpu-usage: 499.642m
                        forecast-arima-memory-usage: 0.446Mi
                        mean-cpu-usage: 499.640m
                        mean-memory-usage: 0.450Mi
                        std-cpu-usage: 0.000m
                        std-memory-usage: 0.000Mi
Selector:               app=redis-leader
Replicas:               2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=redis-leader
           role=leader
           tier=backend
```

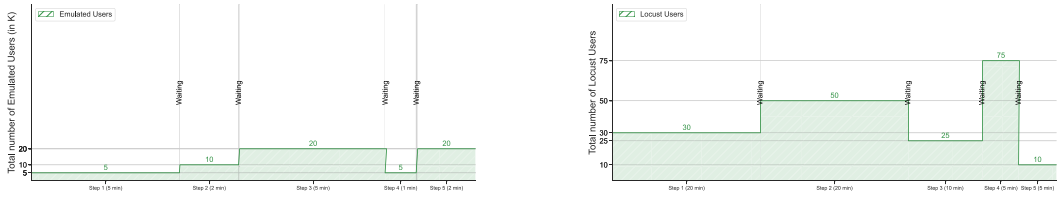
Fig. 8. Illustration of the annotations performed by the *Gwydion-estimator* pod for the *leader* deployment.

6.2. Resource estimation in Kubernetes (K8s)

To create a reliable and scalable manner of forecasting future resource consumption, a container named *Gwydion-estimator* has been developed to enrich K8s deployments with annotations that offer insights into their expected resource utilization. Fig. 7 provides an overview of the *Gwydion-estimator* developed for the *Gwydion* framework, illustrating its primary operations and its interactions with typical K8s components. After deploying the microservice-based application to which *Gwydion* will perform auto-scaling, several monitoring pods are strategically deployed in the cluster to monitor all microservices ①. Each *Gwydion-estimator* pod is responsible for monitoring a specific microservice. These *Gwydion-estimator* pods retrieve current CPU and memory utilization of the given microservice instances by relying on Prometheus (Turnbull, 2018) deployed in the K8s cluster ②. The goal is to collect multiple CPU and memory samples based at regular intervals (e.g., every 15 s) and store them in dedicated CSV files. This data samples serve as the basis for subsequent predictions based on a particular algorithm such as ARIMA ③. Once these predictions are computed in the *Gwydion-estimator*, the K8s deployments are enriched with appropriate annotations, as shown in Fig. 8. These metrics can be included in the observation space of *Gwydion* as a potential indication of the resource consumption of these microservices in the next time step. These annotations provide valuable estimations that can be integrated into the observation space of the *Gwydion* framework. This integration allows *Gwydion* to consider these metrics as potential indicators of the resource consumption of these microservices in the subsequent time steps. This enhanced understanding of resource usage helps *Gwydion* in auto-scaling decisions, ultimately leading to more efficient resource allocation and management.

Table 5
Deployment properties of the evaluated microservice applications.

Application	Deployment	CPU R/L (in m)	RAM R/L (in Mi)	Min/Max Rep. (α_d)
Redis Cluster (a_1)	*Leader* *Follower*	250/500	250/500	1/8
Online Boutique (a_2)	*Frontend*	100/200	64/128	1/8
	Cart	200/300	180/300	
	Product	100/200	64/128	
	Currency	100/200	64/128	
	Payment	100/200	64/128	
	Shipping	100/200	64/128	
	Email	100/200	64/128	
	Checkout	100/200	64/128	
	Recommend.	100/200	64/128	
	Ad	200/300	180/300	
	Redis-cart	70/125	200/256	



(a) Redis Cluster (RC) application.

(b) Online Boutique (OB) application.

Fig. 9. Illustration of the specific workload pattern applied during testing for both applications. A 15-min pattern has been applied for RC, and a 1-hour pattern for OB. If the testing of the algorithm is not completed within the designated time frame, the pattern repeats consecutively until completion.

7. Evaluation setup

This section presents the applications used in the evaluation, followed by the experimental setup.

7.1. Applications

Table 5 shows the deployment properties for the evaluated applications. Different resource requests and limits (i.e., CPU and memory) have been specified for each microservice. The minimum and maximum replication factors are set to 1 and 8, respectively.

Redis Cluster (RC) The first scenario (Fig. 2(a)) relates to the deployment of the RC application (Redis, 2024) consisting of two K8s deployments: *leader* and *follower*. The Redis-benchmark utility (Redis, 2022) is used to generate database queries from emulated clients during the training and testing of the RL agents. The number of emulated clients (i.e., from 0 to 50K users) and the types of queries executed varies dynamically during training, generating different load patterns on the RC application. The goal is that RL agents learn to adapt the allocation scheme according to the current demand. RC is a highly-available application, so auto-scaling mechanisms should ensure no downtime during scaling operations. During testing, a specific workload pattern has been applied to compare the different reward strategies after training as shown in Fig. 9(a). The Redis Exporter (Exporter, 2022) developed for Prometheus is deployed in the K8s cluster to extract metrics regarding the performance of the RC application. The latency for RC (Ψ_{a_1}) corresponds to the calculation of the average response time of the Redis server by collecting the total query duration and the total query response time during the last five minutes as shown in (7). The latency threshold (τ_{a_1}) is set to 250 ms.

$$\Psi_{a_1} = \frac{redis\_commands\_duration\_seconds\_total[5\ m]}{redis\_commands\_processed\_total[5\ m]} \tag{7}$$

Online Boutique (OB) The second scenario (Fig. 2(b)) relates to the OB application (Boutique, 2024) consisting of 11 K8s deployments. It is a web-based e-commerce application where users can browse items and add them to their cart to purchase them. OB represents a microservice application with numerous dependencies between its microservices.

Table 6
The Hardware Configuration of theK8s cluster.

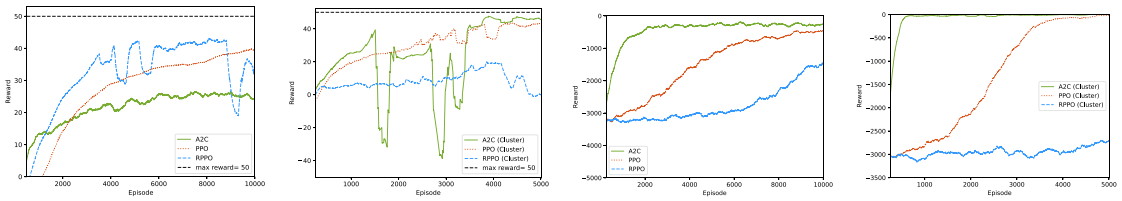
Node	CPU	Memory
Master Worker 1 Worker 2	Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz	48 GB

The *frontend* service receives HTTP requests and forwards them to several services, including *currency* and *product-catalog*. In the evaluation, a load generator based on the locust load tool (Locust, 2021) sends several *GET* and *POST* requests from emulated users (i.e., from 0 to 100 users). During training, the load on the OB application varies dynamically depending on the number of emulated users and the types of requests executed. During testing, a specific pattern has been applied to compare the different reward strategies after training as shown in Fig. 9(b). The Locust Exporter (Solutions, 2022) developed for Prometheus has been deployed in the K8s cluster to collect the average response time of several requests. The latency for OB (Ψ_{a_2}) corresponds to the average response time based on the *GET /cart* request as shown in (8). This request represents a critical user interaction point within the OB's functionality. While also having explored the average response time of various requests, our experiments revealed similar results. Consequently, we have opted to consider only the response time of the *GET /cart* request as it provides a comprehensive understanding of the user experience, while minimizing the number of API calls to Prometheus. The latency threshold (τ_{a_2}) is set to 3 s.

$$\Psi_{a_2} = locust\_avg\_response\_time\_GET\_cart \tag{8}$$

7.2. Testbed implementation

The *Gwydion* framework has been implemented in Python to ease the interaction with both the OpenAI Gym and stable baselines 3 libraries. The K8s Python Client has been used to access a K8s cluster and retrieve information from the given deployments. A K8s-based infrastructure composed of a master node and two worker nodes has



(a) Cost (Simulation). (b) Cost (Cluster). (c) Latency (Simulation). (d) Latency (Cluster).

Fig. 10. Accumulated rewards during training for the Redis Cluster (RC) application.

Table 7
Software Versions of the Testbed.

Software	Version
Python &K8s Python Client	3.11 & 28.1.0
gym & stable baselines 3	0.26.2 & 2.1.0
Kubeadm & Kubectl	v1.26.1
Container Runtime	containerd://1.6.12
Linux Kernel	5.4.0-33-generic
Operating System	Ubuntu 20.04.2 LTS

Table 8
The *Gwydion* environment configuration.

Application	Action Space	Obs. Space
RC	*MultiDiscrete(2, 15)*	7 states
OB	*MultiDiscrete(11, 15)*	34 states

been applied in the evaluation to have reliable and highly available computing resources. The K8s cluster has been set up in the imec Virtual Wall (VWall) infrastructure (Wall, 2023) at IDLab, Belgium. Table 6 details the hardware configurations of each cluster node, and Table 7 lists the software versions applied. In addition, Table 8 shows the environment configurations based on the evaluated applications. In the evaluation, an episode consists of 25 steps where the agent attempts to maximize the reward based on the current demand. For the simulation, the agents 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:

- **Execution Time** of each agent per episode.
- **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.
- **Deployment costs** based on the average number of deployed pods during each episode.
- **Expected latency** based on the average application response time during each episode.

8. Results

This section presents the results obtained for both applications in the evaluation. Section 8.1 accesses the performance of the RL algorithms integrated in *Gwydion* and Section 8.2 quantifies the prediction algorithms supported by *Gwydion* based on different metrics. Moreover, Section 8.3 showcases the last step in our evaluation which relates to the integration of the *Gwydion-estimator* into the RL observation space. Lastly, Section 8.4 summarizes this section by highlighting the insights obtained in this study.

8.1. Reinforcement learning (RL) assessment

Execution Time Table 9 shows the execution time per episode (i.e., 25 steps) during training for all RL algorithms based on the simulation and cluster modes. The simulation mode is significantly faster since the observations come from CSV files rather than retrieved from the K8s API and Prometheus API in the cluster mode. The RL agents have been trained for 250K steps (i.e., 10000 episodes) and 65K/125K steps (i.e., 2500/5000 episodes) for the simulation and cluster mode, respectively. The results show that training RL agents in a real cluster is significantly costly. For the RC application, each episode lasts 70 to 90 s on average in the cluster mode and takes only 0.4 to 3.6 s in the simulation mode, with RPPO being slower than A2C and PPO. All algorithms run even slower for the OB application since it consists of eleven microservices, which makes the observation space significantly larger. In addition, the required execution time for 2500 episodes is shown, highlighting that the simulation needs a few minutes while the cluster more requires a few days to run the exact number of episodes.

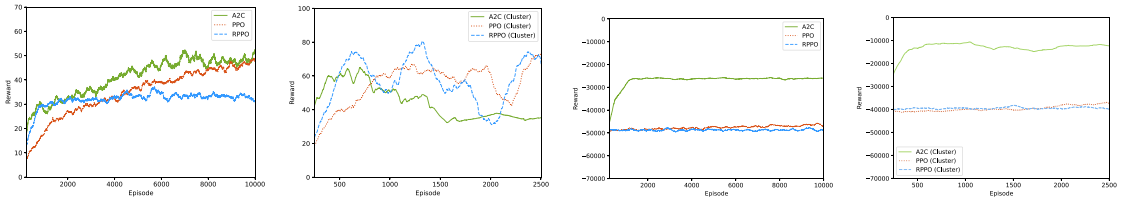
Training Fig. 10 illustrates the accumulated reward for both reward functions during training for the RC application in the simulation and cluster modes. The RL algorithms for the cost function in the cluster mode achieve slightly higher rewards than the trained agents in the simulation mode. However, the training in the cluster mode is more unstable since spikes are obtained for most algorithms. These spikes can be related to unstable API calls or incorrect metrics retrieved from *Prometheus*. Though *Gwydion* supports API retries to mitigate the impact of these erroneous calls on the RL performance. Regarding the latency-aware function, the algorithms explore the action space to find actions that lead to null rewards since it means that the latency is close to zero. Fig. 11 illustrates the accumulated reward for both reward functions during training for the OB application in the simulation and cluster modes. A similar pattern occurs compared to the RC application, where the cluster mode achieves slightly higher accumulated rewards than the simulation mode.

Testing After training the RL algorithms, the saved model configurations (i.e., after 5K or 10K steps) have been executed for 100 episodes to assess the performance of the RL agents with a different demand pattern. KHPA has also been evaluated by enabling it for each deployment in the considered applications. KHPA stands as the most widely adopted baseline for container auto-scaling. Each episode for KHPA consists of the average execution time for each application previously shown in Table 9, where the number of deployed pods and the application's latency are retrieved from *Prometheus*. Figs. 12 and 13 show the results acquired during the testing phase for the KHPA and the various RL algorithms, demonstrating the average number of deployed pods and the average application latency. Most cost-aware algorithms can reduce the number of pods deployed for the RC application compared to KHPA, leading to a slightly higher latency but decreased resource consumption (i.e., CPU and memory usage). A2C (Cluster) deploys an average of 2.0 pods, and PPO (Cluster) deploys 2.19 pods, while KHPA deploys an average of 8.81 pods. In contrast, RPPO (Cluster) deploys significantly more pods than KHPA (10.14 pods). While KHPA achieves an average response time of 50 μs, most RL algorithms achieve an average response time ranging from 300 μs to 6 ms. For the cost-aware strategy (Fig. 12(c)), A2C (Cluster) achieves one of the lowest response times, 0.12 ms on average, with PPO (Cluster) slightly higher at 0.55 ms, and RPPO (Cluster) at 0.08 ms. In the simulation mode, A2C (Simulation) achieves a response time of 2.26 ms, PPO (Simulation) at 2.55 ms, and RPPO (Simulation) at 1.43 ms. For the latency-aware strategy (Fig. 12(d)), PPO (Cluster)

Table 9
The execution time during training for both applications. The cluster mode is significantly more expensive than simulations.

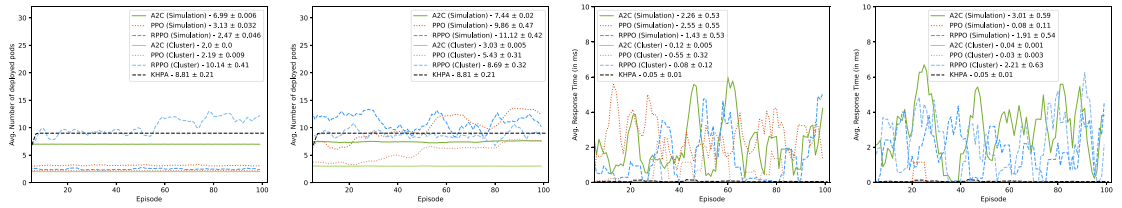
Algorithm	Application	Mode	Execution Time per episode (in s)	Execution Time for 2500 episodes
A2C	RC	S	0.41 ± 0.16	17.08 min
PPO			0.35 ± 0.10	14.58 min
RPPO			3.63 ± 7.28	151.25 min
A2C	RC	C	71.11 ± 48.16	2.06 days (49.38 h)
PPO			73.67 ± 33.78	2.13 days (51.16 h)
RPPO			112.73 ± 19.30	3.26 days (78.28 h)
A2C	OB	S	1.08 ± 0.42	45.0 min
PPO			1.11 ± 0.36	46.25 min
RPPO			4.55 ± 7.20	189.58 min
A2C	OB	C	116.88 ± 17.67	3.38 days (81.17 h)
PPO			117.29 ± 18.74	3.39 days (81.45 h)
RPPO			127.36 ± 27.84	3.68 days (88.44 h)

Mode: S = Simulation, C = Cluster.
Application: RC = Redis Cluster, OB = Online Boutique.



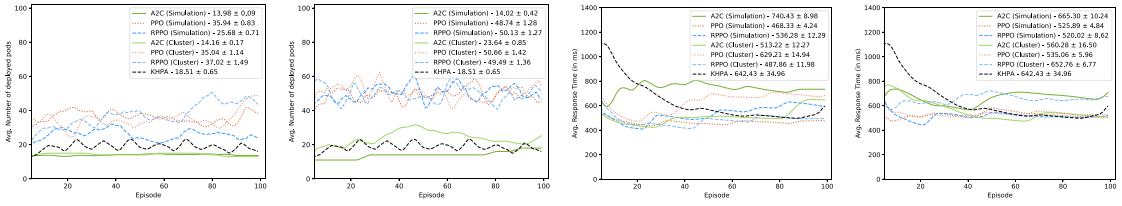
(a) Cost (Simulation). (b) Cost (Cluster). (c) Latency (Simulation). (d) Latency (Cluster).

Fig. 11. Accumulated rewards during training for the Online Boutique (OB) application.



(a) Cost-aware - Pods. (b) Latency-aware - Pods. (c) Cost-aware - Latency. (d) Latency-aware - Latency.

Fig. 12. The testing results for the Redis Cluster (RC) application.



(a) Cost-aware - Pods. (b) Latency-aware - Pods. (c) Cost-aware - Latency. (d) Latency-aware - Latency.

Fig. 13. The testing results for the Online Boutique (OB) application.

achieves the lowest latency at 0.03 ms, followed by A2C (Cluster) at 0.04 ms. RPPO (Cluster), however, performs worse with an average latency of 2.21 ms, while KHPA maintains a consistent 0.05 ms response time. Also, most latency-aware algorithms often deploy additional pods in the cluster to further reduce the expected latency, achieving similar results to KHPA.

KHPA cannot find appropriate scaling actions since it does not consider microservice inter-dependencies, and its stabilization window does not allow fast reactions to sudden demands. The deployment or termination of pod instances by KHPA typically occurs a few seconds after detecting fluctuations in demand. For the RC application, adjustments are faster since only two KHPAs are deployed, whereas for the OB application, around 20 episodes are necessary for KHPA to significantly reduce the latency since OB requires eleven KHPAs (Figs. 13(c) and 13(d)). KHPA obtains an average response time of 642.43 ms, and deploys an average of 18.51 pods, indicating a balanced performance between response time and resource utilization. All RL algorithms exhibit a notable reduction in response time for OB since all microservices are considered in the observation space, enabling the sequential triggering of actions across different deployments. In addition, the proposed cost-aware function prioritizes CPU and memory optimization, while KHPA focuses mainly on CPU usage to trigger scaling actions. Thus, the trained algorithms typically do not scale up microservices when the memory consumption is relatively low, even if CPU is slightly higher. Actually, these extra scaling actions triggered by KHPA have a relatively low benefit on application performance, as

shown by our results. Thus, these extra scaling actions could have been avoided to save computing resources at a slightly higher application response time. In terms of deployment costs for the OB application, it is noteworthy that only A2C outperformed KHPA, achieving a nearly 30% reduction in costs while maintaining consistently lower latency throughout the experiment. For the cost-aware strategy, A2C (Cluster) achieves an average response time of 513.22 ms while deploying an average of 14.16 pods. In comparison, both PPO (Cluster) and RPPO (Cluster) deploy a significantly higher number of pods (35 to 37), resulting in low latency, ranging from 487 ms to 629 ms. While their latency is competitive, the higher number of pods suggests a trade-off between latency and resource efficiency. When it comes to latency-aware functions for the OB application, most RL algorithms achieve lower latency compared to KHPA by scaling up the number of pods, typically ranging from 14 to 50 pods on average. However, A2C stands out for obtaining a balanced trade-off, achieving low latency without requiring an excessively high number of pods. A2C (Cluster) deploys more pods (23.64) compared to A2C (Simulation) (14.02), leading to reduced latency, with A2C achieving 560.28 ms in cluster mode versus 665.30 ms in simulation. In contrast, PPO and RPPO maintain consistent pod deployment numbers and latency between simulation and cluster modes.

The performance fluctuations of the RL algorithms are influenced by several factors inherent to the dynamic nature of microservice environments and the learning process of each RL algorithm:

- **Exploration–Exploitation Trade-off:** A2C, PPO, and RPPO continuously balance between exploring new scaling strategies and exploiting known optimal actions. This balance can result in periodic fluctuations as the algorithms experiment with different actions to find the best scaling policy.
- **Learning Rate and Policy Updates:** The frequency of policy updates in RL algorithms can influence the stability of their decisions. Larger or more frequent updates may lead to more noticeable fluctuations in performance as the algorithms adjust to new policies.
- **Workload Variability:** The algorithms are exposed to varying demand patterns depending on whether these are running in simulation or cluster modes. These fluctuations in workload can cause temporary increases in response time and resource usage as the RL algorithms adapt to the changing environment.
- **Microservice Inter-dependencies:** The interdependent nature of microservices means that changes in one service can affect others. As the RL algorithms learn to optimize the entire application, adjustments in scaling for one microservice may temporarily disrupt the performance of others, contributing to the observed fluctuations.

In conclusion, the performance differences observed in this work emphasize the intricate balance between latency reduction and resource efficiency in microservice environments. While all RL algorithms show improvements over KHPA in various aspects, A2C demonstrates the most balanced performance across both cost-aware and latency-aware scenarios. PPO and RPPO, while achieving low latency, do so at the expense of an increased number of pods, which may not be suitable for every use case. In addition, the discrepancies between simulation and cluster results highlight the importance of testing in real environments to validate and refine the models developed in simulations.

8.2. Gwydion's prediction assessment

The performance of the statistical and ML-based algorithms has been assessed based on the collected datasets.[4] for the simulation mode of *Gwydion* These algorithms have been evaluated based on the following metrics:

- **Root Mean Square Error (RMSE).**
- **Mean Absolute Error (MAE).**
- **Mean Absolute Percentage Deviation (MAPD).**

RMSE, MAE, and MAPD are typical metrics used to evaluate the performance of predictive models. RMSE measures the square root of the average of the squared differences between predicted and actual values, giving more importance to larger errors. MAE computes the average of the absolute differences between predicted and actual values, treating all errors equally. MAPD calculates the average percentage difference between predicted and actual values, providing insights into the relative error. RMSE is typically sensitive to outliers, while MAE is more robust, and MAPD provides error information as a percentage of actual values, useful for comparing models across different datasets or when the scale of the data varies. All metrics have been used extensively to assess the accuracy of predictions.

Table 10 presents the achieved results for all deployments in the evaluated applications. The Naive algorithm performed significantly well since CPU usage values are typically collected every 1–3 s with minimal variance between successive collected values. In these scenarios, Naive proves to be a robust estimator since small differences exist between the current CPU usage and the subsequent sample. However, its effectiveness decreases if the samples are aggregated or acquired less frequently, such as every 15–30 s. In these cases, the variability between consecutive values increases, leading to decreased performance of the Naive algorithm. Also, ARIMA demonstrated comparable performance to Naive, consistently yielding lower RMSE values across various microservices. SES often showed inferior performance compared to Naive and ARIMA, particularly in scenarios where the data has less clear patterns or trends. Despite its superior ability to model complex patterns, LSTM comes with higher computational and memory demands due to the need to train and maintain the neural network. However, LSTM showed lower performance compared to simpler models, though it can capture intricate usage patterns effectively. For example, LSTM performed well for the *Ad* microservice in which sudden changes occur in its CPU usage.

In summary, while Naive and ARIMA generally provided strong results with lower complexity, LSTM was particularly effective for handling sudden changes but at the cost of increased computational resources. SES, while adding moderate complexity, often fell short compared to Naive and ARIMA. The choice of the prediction algorithm in *Gwydion* thus depends on the specific workload characteristics and the trade-offs between accuracy and resource consumption.

8.3. Integration of Gwydion's prediction into RL

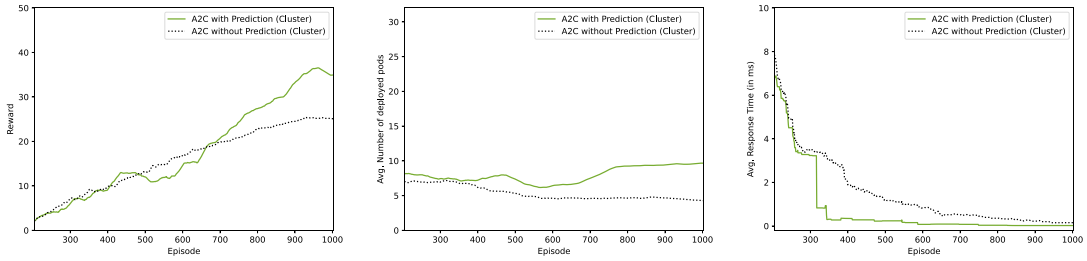
As a final evaluation step, the *Gwydion-estimator* has been integrated into the RL algorithms by incorporating CPU and memory forecast predictions into the observation space. The purpose is to assess the impact on the performance of *Gwydion* when agents are provided with resource estimations. Fig. 14 shows the training results for the RC application in the cluster mode for the A2C algorithm, which already achieves notable performance even without resource estimations. The inclusion of resource estimations enables the agent to achieve higher rewards at a faster rate compared to training without them. The A2C algorithm attains accumulated rewards exceeding 30 after just 800 episodes. This indicates a significant reduction in average response time from 6 ms to 0.7 ms since the beginning of training, while maintaining low deployment costs to meet current demand. These findings demonstrate the potential of integrating prediction capabilities into the RL observation space to advance efficient auto-scaling mechanisms in future cloud environments.

[4] Example of collected datasets: https://zenodo.org/records/7944661.

Table 10
Performance comparison of different prediction algorithms for the CPU usage estimation.

Redis Cluster (RC) application												
Dep.	Naive			SES			LSTM			ARIMA		
	R	M	P	R	M	P	R	M	P	R	M	P
Leader	11.83	**2.49**	**1.07**	16.11	5.87	2.68	15.63	8.21	5.09	**11.78**	2.68	1.15
Follower	3.49	**0.47**	**1.61**	5.04	1.75	5.85	6.60	4.53	17.81	**3.46**	0.56	1.98
Online Boutique (OB) application												
	Naive			SES			LSTM			ARIMA		
Dep.	R	M	P	R	M	P	R	M	P	R	M	P
Frontend	9.98	**1.64**	**0.48**	12.10	3.56	1.05	18.53	7.91	2.35	**9.43**	2.25	0.67
Cart	25.90	**10.12**	**5.69**	34.01	18.19	11.24	27.16	16.43	10.2	**25.72**	10.76	6.28
Product	**13.15**	**4.84**	**1.63**	16.49	8.59	2.96	22.95	13.90	4.93	13.23	4.92	1.66
Currency	18.08	**6.84**	**2.27**	23.35	11.68	3.93	28.23	16.06	5.59	**17.26**	7.83	2.67
Payment	19.24	**7.17**	**10.35**	24.6	11.79	18.11	25.93	19.17	27.26	**18.61**	7.78	12.04
Shipping	8.53	**3.58**	**3.90**	10.96	6.15	6.99	14.58	9.87	12.46	**8.50**	3.73	4.14
Email	15.74	**6.04**	**9.02**	20.64	10.33	16.91	17.56	9.20	15.01	**15.63**	6.29	9.63
Checkout	**12.54**	**5.86**	**4.07**	15.23	10.24	7.24	19.38	14.63	10.76	12.57	5.87	**4.07**
Recomm.	**21.97**	**7.63**	**2.83**	27.25	13.45	5.15	26.84	15.52	5.95	22.03	7.83	2.93
Ad	137.74	**30.15**	**6.98**	210.58	66.26	16.44	**125.54**	136.86	136.27	131.93	33.66	8.67
Redis-cart	**1.46**	**0.39**	**2.57**	2.28	1.57	8.98	5.35	4.19	20.85	**1.46**	0.42	2.77

R = RMSE, **M = MAE**, **P = MAPD**.



(a) Accumulated Reward. (b) Deployed Pods. (c) Latency (in ms).

Fig. 14. The training results for the Redis Cluster application when prediction is integrated into the RL observation space.

8.4. Summary

Our results show that microservice inter-dependencies play a pivotal role in the efficient auto-scaling of microservice applications. Two opposing scaling strategies have demonstrated that RL algorithms can find appropriate actions by considering microservice inter-dependencies when offered reliable information in their observation space. During training, the cluster mode is a more reliable scenario for training RL agents, but it is a very costly operation. The cluster mode is on average 100 times more expensive than the simulation mode. In fact, during the testing phase, the simulation mode performed similarly to the cluster mode, demonstrating RL agents can be trained offline with simulation-based environments and then validated in real operational infrastructures. A2C seems more suitable for generalization since it achieved higher rewards than both PPO and RPPO during different demand patterns.

The prediction algorithms integrated in *Gwydion* are designed to assist RL agents in estimating future resource consumption, thereby triggering proactive actions. By integrating predictions into the RL observation space, the RL learning process is faster, and the achieved performance is higher by leveraging on resource forecasts. ARIMA has emerged as a viable option for estimating resource usage within K8s environments, and Naive as a reliable benchmark for prediction methods, particularly effective when minimal variance is present between sequentially collected values.

In conclusion, the main contribution of this paper is the design and implementation of generic interfaces between OpenAI Gym and K8s clusters, allowing a seamless experiment for RL training on real cloud environments. This work provides a benchmark framework for RL-based auto-scaling research in K8s of interest for both the cloud and network management communities.

9. Open challenges & future directions

This section discusses remaining hurdles on auto-scaling in modern cloud platforms not yet fully addressed by literature and highlights future directions. Containers have revolutionized application deployment, offering high flexibility and enhancements in scheduling and scaling features. However, this transition is accompanied by multiple challenges related to efficient allocation and scaling, making it essential to address them to ensure the sustainability and cost-effectiveness of container-based cloud infrastructures.

Composite Metrics for Auto-scaling will help to create unified models that consider various metrics simultaneously, such as computing resources, latency, and error rates. This multidimensional evaluation provides a more comprehensive view of the performance of the application. An organization might prefer to optimize processing times and cost reduction, while other prefers to provide fast responses to their users. This work assessed two opposing scaling strategies: cost-aware and latency-aware. As future work, advanced ML models can be used to learn how to generate an optimal composite score. Composite metrics for scaling decisions will represent a more holistic approach to auto-scaling, considering the multifaceted nature of microservice performance.

Cold Start Reduction is still a major challenge today in modern cloud environments. A potential research direction is to explore predictive models that anticipate when cold starts are likely to occur. By analyzing historical patterns and expected traffic demands, auto-scaling mechanisms can proactively warm up microservice instances before they are actually needed. It will minimize the impact of cold starts on user experience. Moreover, in serverless computing (Li et al., 2023), functions with similar resource requirements can be clustered

together. When one function within a cluster is invoked, it warms up the entire cluster. A potential research direction is to study how to optimize the clustering of these functions to minimize cold starts and resource wastage. Additionally, for complex microservice-based applications, deploying and initializing certain microservice instances in advance will be beneficial, allowing them to warm up gradually before becoming fully operational. Reducing cold starts is crucial to ensure that microservices can provide reliable and responsive services. As microservice architectures evolve, reducing cold starts is essential to maintain an enhanced user experience, especially in containerized and serverless environments.

Sustainable Auto-scaling practices should be a major direction in the next few years. Future research should investigate scaling policies that consider the environmental impact, including scaling decisions that align with renewable energy sources and data center energy efficiency. Auto-scaling strategies that optimize data transfers and minimize network latency to conserve energy and reduce carbon emissions. Also, the design and development of methodologies for tracking the carbon footprint of auto-scaling decisions can enable organizations to make more environmentally responsible choices. Moreover, exploring the adoption of energy-efficient hardware in data centers, such as low-power processors and other hardware components, could help consume less power while maintaining high performance levels.

Multi-cloud deployments will be of paramount importance in the next years. Future research should investigate mechanisms for auto-scaling that abstract away cloud-specific details, enabling seamless scaling across multiple cloud providers and on-premise environments. It will enable applications to run on any cloud platform without significant modifications, ensuring high portability and flexibility in adopting cloud providers. Developing standards and best practices for interoperable auto-scaling solutions across distinct cloud providers can simplify multi-cloud and hybrid deployments, facilitating workload distribution and data exchange. Multi-cloud deployments will offer numerous benefits, but also introduce complexities that require careful planning and management. Further research will help organizations harness the advantages of multi-cloud and hybrid strategies while effectively addressing open challenges and optimizing their cloud infrastructure.

10. Conclusions

This paper presents the *Gwydion* framework inspired by the OpenAI Gym library and the widely adopted K8s platform, enabling the creation of suitable RL environments for auto-scaling research. The framework has been released in open-source, allowing researchers to evaluate their auto-scaling techniques. This paper extensively studies microservice inter-dependencies in auto-scaling since current applications represent complex graphs of loosely-coupled microservices, making auto-scaling a significantly challenging assignment. *Gwydion* builds further on our previous *gym-hpa* framework by refining the RL design and integrating prediction algorithms to estimate the resource consumption of microservices. The goal is to improve resource usage and reduce the application's response time in future clouds by applying RL for proper horizontal scaling of microservice applications with complex inter-dependencies. Experiments with microservice benchmark applications, such as RC and OB, showed that *Gwydion* can significantly decrease deployment costs or the application latency compared to default auto-scaling mechanisms. For RC, cost-aware algorithms can reduce the number of deployed pods (2 to 4), resulting in slightly higher latency (300 µs to 6 ms) but lower resource consumption. For OB, all RL algorithms exhibit a notable response time improvement by considering all microservices in the observation space, enabling the sequential triggering of actions across different deployments. This leads to nearly 30% cost savings while maintaining consistently lower latency throughout the experiment. In addition, RL training within real K8s clusters is significantly more expensive compared to simulation-based environments, needing on average 100x its execution time. As future work, novel descheduling policies will be studied to terminate underperforming microservices while running *Gwydion's* scaling strategies.

CRediT authorship contribution statement

José Santos: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Efstratios Reppas:** Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Tim Wauters:** Writing – review & editing, Supervision, Resources, Project administration, Methodology, Investigation. **Bruno Volckaert:** Writing – review & editing, Supervision, Resources, Project administration. **Filip De Turck:** Writing – review & editing, Supervision, Resources, Project administration.

Declaration of competing interest

The authors 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

José Santos is funded by the Research Foundation Flanders (FWO), Belgium, grant number 1299323N. This work is supported by the Belgian Chancellery of the Prime Minister (Grant: AIDE-BOSA).

Data availability

The framework has been released in open-source.

References

Amazon, 2024a. Amazon elastic container service (Amazon ECS). [Online]. Available: https://aws.amazon.com/ecs/. (Accessed on 13 September 2024).

Amazon, 2024b. Amazon elastic kubernetes service. [Online]. Available: https://aws.amazon.com/eks/. (Accessed on 13 February 2024).

Arcaini, P., Mirandola, R., Riccobene, E., Scandurra, P., 2020. Model-based testing for MAPE-K adaptation control loops. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, 7, IEEE, pp. 43–51. http://dx.doi.org/10.1109/icstw50294.2020.00024.

AWS, 2024. Service auto scaling. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest//developerguide/service-auto-scaling.html. (Accessed on 13 September 2024).

Boutique, O., 2024. Online boutique, a cloud-native microservices demo application. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo. (Accessed on 13 September 2024).

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W., 2016. Openai gym. http://dx.doi.org/10.48550/ARXIV.1606.01540.

Burns, B., Beda, J., Hightower, K., 2019. Kubernetes: Up and Running: Dive into the Future of Infrastructure. O'Reilly Media.

Burns, B., Beda, J., Hightower, K., Evenson, L., 2022. Kubernetes: Up and Running. O'Reilly Media, Inc..

Calheiros, R.N., Masoumi, E., Ranjan, R., Buyya, R., 2015. Workload prediction using ARIMA model and its impact on cloud applications' QoS. IEEE Trans. Cloud Comput. 3 (4), 449–458. http://dx.doi.org/10.1109/tcc.2014.2350475.

Danilo, A., Michele, C., Lancellotti, R., Michele, G., 2021. A hierarchical receding horizon algorithm for qos-driven control of multi-iaas applications. IEEE Trans. Cloud Comput. 9 (2), 418–434. http://dx.doi.org/10.1109/tcc.2018.2875443.

Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L., Madry, A., 2019. Implementation matters in deep rl: A case study on ppo and trpo. In: International Conference on Learning Representations.

Exporter, P., 2022. Prometheus redis metrics exporter. [Online]. Available: https://github.com/oliver006/redis_exporter. (Accessed on 28 August 2022).

Farokhi, S., Jamshidi, P., Lakew, E.B., Brandic, I., Elmroth, E., 2016. A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach. Future Gener. Comput. Syst. 65, 57–72. http://dx.doi.org/10.1016/j.future.2016.05.028.

Gergin, I., Simmons, B., Litoiu, M., 2014. A decentralized autonomic architecture for performance control in the cloud. In: 2014 IEEE International Conference on Cloud Engineering. IEEE, http://dx.doi.org/10.1109/ic2e.2014.75.

Giordani, M., Polese, M., Mezzavilla, M., Rangan, S., Zorzi, M., 2020. Toward 6G networks: Use cases and technologies. IEEE Commun. Mag. 58 (3), 55–61. http://dx.doi.org/10.1109/mcom.001.1900411.

Google, 2024. Google kubernetes engine (GKE), the most scalable and fully automated kubernetes service. [Online]. Available: https://cloud.google.com/kubernetes-engine?hl=en. (Accessed on 13 February 2024).

Hat, R., 2024. Red hat OpenShift container platform. [Online]. Available: https://www.redhat.com/en/technologies/cloud-computing/openshift. (Accessed on 13 September 2024).

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., Silver, D., 2018. Rainbow: Combining improvements in deep reinforcement learning. In: Thirty-Second AAAI Conference on Artificial Intelligence.

Janardhanan, D., Barrett, E., 2017. CPU workload forecasting of machines in data centers using LSTM recurrent neural networks and ARIMA models. In: 2017 12th International Conference for Internet Technology and Secured Transactions. ICITST, IEEE, http://dx.doi.org/10.23919/icitst.2017.8356346.

Kubernetes, 2022a. A deployment provides declarative updates for pods and ReplicaSets.. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. (Accessed on 28 July 2022).

Kubernetes, 2022b. Resource management for pods and containers. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/. (Accessed on 28 July 2022).

Kubernetes, 2024a. Horizontal pod autoscaler. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. (Accessed on 13 September 2024).

Kubernetes, 2024b. Vertical pod autoscaler. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler. (Accessed on 13 September 2024).

Kumar, J., Singh, A.K., 2018. Workload prediction in cloud using artificial neural network and adaptive differential evolution. Future Gener. Comput. Syst. 81, 41–52. http://dx.doi.org/10.1016/j.future.2017.10.047.

Lee, D., Yoo, J.-H., Hong, J.W.-K., 2020. Deep Q-networks based auto-scaling for service function chaining. In: 2020 16th International Conference on Network and Service Management. CNSM, IEEE, http://dx.doi.org/10.23919/cnsm50824.2020.9269107.

Li, Y., Lin, Y., Wang, Y., Ye, K., Xu, C., 2023. Serverless computing: State-of-the-art, challenges and opportunities. IEEE Trans. Serv. Comput. 16 (2), 1522–1539. http://dx.doi.org/10.1109/tsc.2022.3166553.

Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., Babar, M.A., 2021. Understanding and addressing quality attributes of microservices architecture: A systematic literature review. Inf. Softw. Technol. 131, 106449. http://dx.doi.org/10.1016/j.infsof.2020.106449.

Locust, 2021. An open source load testing tool. [Online]. Available: https://locust.io/. (Accessed on 2 December 2021).

Medel, V., Tolosana-Calasanz, R., Bañares, J.Á., Arronategui, U., Rana, O.F., 2018. Characterising resource management performance in kubernetes. Comput. Electr. Eng. 68, 286–297. http://dx.doi.org/10.1016/j.compeleceng.2018.03.041.

Messias, V.R., Estrella, J.C., Ehlers, R., Santana, M.J., Santana, R.C., Reiff-Marganiec, S., 2015. Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure. Neural Comput. Appl. 27 (8), 2383–2406. http://dx.doi.org/10.1007/s00521-015-2133-3.

Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K., 2016. Asynchronous methods for deep reinforcement learning. In: International Conference on Machine Learning. PMLR, pp. 1928–1937.

Newman, S., 2021. Building Microservices. O'Reilly Media, Inc..

Nouri, S.M.R., Li, H., Venugopal, S., Guo, W., He, M., Tian, W., 2019. Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications. Future Gener. Comput. Syst. 94, 765–780. http://dx.doi.org/10.1016/j.future.2018.11.049.

openAIGym, 2024. The action spaces in openAI gym. [Online]. Available: https://github.com/openai/gym/tree/master/gym/spaces. (Accessed on 13 September 2024).

Ostertagova, E., Ostertag, O., 2012. Forecasting using simple exponential smoothing method. Acta Electrotech. Inform. 12 (3), http://dx.doi.org/10.2478/v10198-012-0034-2.

Qu, C., Calheiros, R.N., Buyya, R., 2018. Auto-scaling web applications in clouds: A taxonomy and survey. ACM Comput. Surv. 51 (4), 1–33. http://dx.doi.org/10.1145/3148149.

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

Rao, S.N., Shobha, G., Prabhu, S., Deepamala, N., 2019. Time series forecasting methods suitable for prediction of CPU usage. In: 2019 4th International Conference on Computational Systems and Information Technology for Sustainable Solution. CSITSS, vol. 9, IEEE, pp. 1–5. http://dx.doi.org/10.1109/csitss47250.2019.9031015.

Ratcliffe, D.S., Hofmann, K., Devlin, S., 2019. Win or learn fast proximal policy optimisation. In: 2019 IEEE Conference on Games (CoG), vol. 23, IEEE, pp. 1–4. http://dx.doi.org/10.1109/cig.2019.8848100.

Rattihalli, G., Govindaraju, M., Lu, H., Tiwari, D., 2019. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In: 2019 IEEE 12th International Conference on Cloud Computing. CLOUD, IEEE, http://dx.doi.org/10.1109/cloud.2019.00018.

Redis, 2022. How fast is redis?. [Online]. Available: https://redis.io/topics/benchmarks. (Accessed on 28 August 2022).

Redis, 2024. Redis, an open source in-memory data structure store. [Online]. Available: https://redis.io/. (Accessed on 13 September 2024).

Rossi, F., Nardelli, M., Cardellini, V., 2019. Horizontal and vertical scaling of container-based applications using reinforcement learning. In: 2019 IEEE 12th International Conference on Cloud Computing. CLOUD, IEEE, http://dx.doi.org/10.1109/cloud.2019.00061.

Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., Strack, B., Witusowski, P., Hand, S., et al., 2020. Autopilot: workload autoscaling at google. In: Proceedings of the Fifteenth European Conference on Computer Systems. EuroSys '20, ACM, http://dx.doi.org/10.1145/3342195.3387524.

Santos, J., Wang, C., Wauters, T., De Turck, F., 2023a. Diktyo: Network-aware scheduling in container-based clouds. IEEE Trans. Netw. Serv. Manag. 20 (4), 4461–4477. http://dx.doi.org/10.1109/tnsm.2023.3271415.

Santos, J., Wauters, T., Volckaert, B., De Turck, F., 2019. Towards network-aware resource provisioning in kubernetes for fog computing applications. In: 2019 IEEE Conference on Network Softwarization. NetSoft, IEEE, http://dx.doi.org/10.1109/netsoft.2019.8806671.

Santos, J., Wauters, T., Volckaert, B., De Turck, F., 2021. Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions. IEEE Commun. Surv. Tutor. 23 (4), 2557–2589. http://dx.doi.org/10.1109/comst.2021.3095358.

Santos, J., Wauters, T., Volckaert, B., De Turck, F., 2023b. 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, http://dx.doi.org/10.1109/noms56928.2023.10154298.

Shumway, R.H., Stoffer, D.S., Shumway, R.H., Stoffer, D.S., 2017. ARIMA models. In: Time Series Analysis and Its Applications: with R Examples. Springer, pp. 75–163.

Singh, P., Gupta, P., Jyoti, K., Nayyar, A., 2019. Research on auto-scaling of web applications in cloud: Survey, trends and future directions. Scalable Comput.: Pract. Exp. 20 (2), 399–432. http://dx.doi.org/10.12694/scpe.v20i2.1537.

Soldani, J., Montesano, G., Brogi, A., 2021. What went wrong? Explaining cascading failures in microservice-based applications. In: Service-Oriented Computing. Springer International Publishing, pp. 133–153. http://dx.doi.org/10.1007/978-3-030-87568-8_9.

Solutions, C., 2022. Locust exporter. [Online]. Available: https://github.com/ContainerSolutions/locust_exporter. (Accessed on 28 August 2022).

Srirama, S.N., Adhikari, M., Paul, S., 2020. Application deployment using containers with auto-scaling for microservices in cloud environment. J. Netw. Comput. Appl. 160, 102629. http://dx.doi.org/10.1016/j.jnca.2020.102629.

Toka, L., Dobreff, G., Fodor, B., Sonkoly, B., 2021. Machine learning-based scaling management for kubernetes edge clusters. IEEE Trans. Netw. Serv. Manag. 18 (1), 958–972. http://dx.doi.org/10.1109/tnsm.2021.3052837.

Toosi, A.N., Son, J., Chi, Q., Buyya, R., 2019. ElasticSFC: Auto-scaling techniques for elastic service function chaining in network functions virtualization-based clouds. J. Syst. Softw. 152, 108–119. http://dx.doi.org/10.1016/j.jss.2019.02.052.

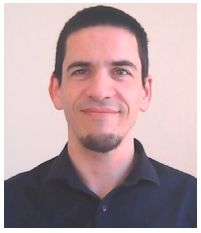
Turnbull, J., 2018. Monitoring with Prometheus. Turnbull Press.

Wall, V., 2023. The virtual wall emulation environment.. [Online]. Available: https://doc.ilabt.imec.be/ilabt/virtualwall/index.html. (Accessed on 28 May 2023).

Yu, Y., Si, X., Hu, C., Zhang, J., 2019. A review of recurrent neural networks: LSTM cells and network architectures. Neural Comput. 31 (7), 1235–1270. http://dx.doi.org/10.1162/neco_a_01199.

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.

Efstratios Reppas is studying toward an Integrated M.Sc. in Electrical and Computer Engineering at the National Technical University of Athens (NTUA). His research interests include algorithms, machine learning, and decision-making systems. He did an internship during July–August 2023 in the Internet Technology and Data Science Lab (IDLab) Research Group at Ghent University - imec, Belgium.



Tim Wauters received the M.Sc. and Ph.D. degrees in electro-technical engineering from Ghent University, in 2001 and 2007, respectively. He has been working as a Postdoctoral Fellow of F.W.O.-V. with the Department of Information Technology (INTEC), Ghent University. He is currently active as a Senior Researcher at imec. His work has been published in more than 170 scientific publications. His research interests include design and management of networked services, covering multimedia distribution, cybersecurity, big data, and smart cities.

Bruno Volckaert is professor advanced software engineering and secure distributed systems in the INTEC department at Ghent University and imec's IDLab group. His current research deals with reliable and high performance distributed software for a.o. scalable data ingestion and processing, scalable cybersecurity intrusion detection and autonomous optimization of cloud-based applications. He has worked on over 65 (inter)national research projects and is (co-)author of over 200 peer-reviewed papers in international journals and conference proceedings.

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.