Università degli Studi di Ferrara

DOCTORAL COURSE IN
ENGINEERING SCIENCE

CYCLE XXXVIII

COORDINATOR Prof. Spina Pier Ruggero

Learning to Orchestrate: AI and Digital Twin Approaches for the Compute Continuum

Scientific/Disciplinary Sector (SDS) ING-INF/05

Ph.D. Candidate
Dott. Zaccarini Mattia

Supervisor
Prof. Tortonesi Mauro

Cosupervisor
Dr. Poltronieri Filippo

Year 2022/2025

Acknowledgements

This thesis is the culmination of my Ph.D. at the University of Ferrara and it would not have been possible without the invaluable contributions of several individuals.

A special thanks to Prof. Mauro Tortonesi from the Department of Mathematics and Computer Science, University of Ferrara, and Dr. Filippo Poltronieri from the Department of Engineering, University of Ferrara, for their guidance and never-ending support.

A special thanks goes to the city of Ghent and to all the wonderful colleagues and people I met during my research stay. Ghent became a home away from home, offering both inspiration and warmth. The kindness, collaboration, and shared moments with the people I met there have left a permanent mark on this journey.

To my family and friends, thank you for standing by me during every high and low of this journey. Your love, your humor, and your ability to remind me of what truly matters have sustained me more than you know. I could not have reached this milestone without you.

In loving memory of my grandmother Maria; Although she passed away during the course of my doctoral studies, her love, resilience, and quiet encouragement shaped who I am and sustained me even in her absence. I carry her with me in every step of this achievement.

Abstract

In recent years, modern computing environments have been evolving towards the *Compute Continuum* (CC), an ecosystem that encloses cloud, fog, and edge resources to support increasingly complex, distributed, and latency-sensitive applications. Within this paradigm, Kubernetes (K8s) has become the de-facto standard for orchestrating microservice-based architectures, although its operational complexity and dynamic nature pose significant challenges to efficient deployment, configuration, and fault resilience.

This doctoral research addresses these challenges by combining Digital Twin (DT) modeling, Reinforcement Learning (RL), and Computational Intelligence (CI) to design data-driven adapted orchestration frameworks for distributed CC environments. The main contribution that this thesis introduces lies in KubeTwin (KT), a DT framework that enables the accurate simulation of K8s deployments, allowing what-if analyses and automated optimization of management policies. KT leverages both simulation-based inference and Machine Learning (ML) techniques, such as Mixture Density Network (MDN), to model microservice response times and validate DT accuracy in realistic multi-access edge scenarios.

Building on this foundation, this research also explores RL-based orchestration for optimal service allocation across multi-cluster K8s setups in the CC. The proposed approaches, implemented in framework solutions such as gym-multi-k8s and HephaestusForge, integrate Deep Sets (DS) neural architectures to generalize deployment policies over varying cluster configurations, achieving near-optimal trade-offs between latency, cost, and fairness. Complementary studies compare multiple optimization algorithms, from Deep Q-Network (DQN) and Proximal Policy Optimization (PPO) to Genetic and swarm-based metaheuristics, highlighting the strengths of hybrid RL–CI strategies and Multi-Objective (MO) optimization for adaptive service management. As a final contribution, this thesis also investigates resilience and fault tolerance through Twin-Enhanced Learning for Kubernetes Applications (TELKA), a DT-driven RL scheduler that incorporates Chaos Engineering (CE) to detect and mitigate system faults proactively.

Overall, this work advances the state of the art in autonomous orchestration for distributed cloud systems by integrating DTs, RL, and CI into several unified method-

ological frameworks. The proposed solutions demonstrate how simulation-enhanced learning and intelligent optimization can provide flexible, efficient, and resilient orchestration strategies for next-generation cloud native applications operating across the CC.

Contents

Chapter 1

Introduction

In recent years, the research community has observed a reshaping of network infrastructures, with a growing interest and adoption of the CC paradigm. CC brings together a variety of computational and storage resources, spreading them across different layers resulting in a unified ecosystem establishment [1]. In this way, users would not only rely on the resources available at the edge but could also exploit computing resources provided by fog or cloud providers. Consequentially, CC opens new possibilities to distribute the load of computationally intensive services such as ML applications, online gaming, Big Data management, and so on [2], [3]. It is important to note that all these new innovative options and the inclusion of a plethora of new devices in the ecosystem creates a large number of potential threads which require the adoption of new effective countermeasures [4]. At the heart of the CC paradigm lies the orchestration challenge, which is related to determining the optimal placement of microservices within the CC, taking into account factors such as task duration, computational demands, and latency requirements. In such a distributed and heterogeneous environment as the one that CC brings to the table, effective service management becomes crucial to ensure seamless service delivery and resource optimization. However, efficiently coordinating resource allocation, exploitation, and management in the CC represents quite a challenge [5]. Applications deployed in these environments must adhere to strict Key Performance Indicators, particularly in terms of latency and resource efficiency [6]. Also, the complexity of this problem is expected to increase with the advent of next-generation networks, such as 6G [7], which promise to unlock novel use cases by offering significantly lower latency and higher bandwidth than 5G [8]. This requires careful strategies for placement, migration, and resource allocation to ensure that the system can dynamically adapt to variations in workload and network conditions. Therefore, there is a need for service fabric management solutions capable of efficiently allocating multiple service components using a limited pool of devices distributed among the CC, also considering the peculiar characteristics of the resources available at each layer. In addition, the significant dynamicity of the CC scenario calls for adaptive/intelligent orchestrators that

can autonomously learn the best allocation of services.

At the same time, the landscape of application deployment and life-cycle management has experienced a profound transformation with the advent of containers [9]. From traditional monolithic structures, applications have evolved into intricate loosely coupled microservices, resulting in substantial improvements in deployment flexibility and operational efficiency [10]. However, efficient management of modern microservice-based applications requires sophisticated orchestration solutions. The rise of innovative paradigms such as Fog Computing [11], [12], Edge Computing [13], [14], and the CC [15], [16] puts even more pressure on popular cloud infrastructures to support novel use cases: highly mobile self-driving vehicles [17], bandwidth-intensive Extended Reality applications, and ultra-reliable Industrial IoT services [18]. These use cases require computing resources closer to devices and end users, but the lack of efficient multi-cluster management features has hindered the deployment of these applications due to their stringent latency and bandwidth requirements [19]. The existing literature focuses mainly on single-cluster scenarios, with a few works that address multi-cluster orchestration [20], [21]. The orchestration dilemma becomes notably more complex in these multi-cluster scenarios: several clusters add a layer of complexity to the overall system due to their unique configurations, resource capacities, and network settings. Ensuring low latency across geographically distributed clusters poses a significant challenge, which is even exacerbated by the complexity of container-based applications, typically composed of multiple microservices, each with different and strict requirements. Therefore, orchestrator solutions for the CC must make decisions regarding the location of the deployment of each microservice, including whether to concentrate instances within a single cluster or distribute them across multiple clusters. This approach helps overcome the limitations of traditional scheduling methods in managing the dynamic changes inherent in microservice-based architectures. To achieve this, an effective strategy is essential for determining when to distribute microservice instances across different clusters focused on optimizing various performance factors such as deployment costs and the application's latency. Moreover, since individual microservices often exhibit varying workloads based on user demand and application complexity, efficient orchestration mechanisms are crucial to prevent the overload of compute nodes and to ensure that computing resources are used efficiently within the CC.

The growing proliferation of Artificial Intelligence (AI) and ML tasks introduces new orchestration challenges across the CC [22]–[24]. These include workloads such as Large Language Models, where even basic operations such as hyperparameter tuning can require huge computational power [25], as well as distributed inference [26]. Although 6G will improve network capabilities, realizing its full potential requires the development of intelligent orchestration mechanisms. These mechanisms must effectively leverage the diversity of resources in multi-cluster environments by considering a wide range of deployment criteria, ranging from available CPU and RAM to pricing models and Quality of Service (QoS) metrics such as round-trip latency and service

availability. Despite the significant effort to address optimization problems in heterogeneous and multi-cluster scenarios [27]–[30], many state-of-the-art automated management solutions (e.g., Amazon EKS [31], Platform 9 [32]) still provide only limited support for simultaneously optimizing cost efficiency and adherence to Service Level Agreements, including availability and latency goals. Moreover, while a few existing solutions acknowledge multiple objectives, these often tend to rely on conventional methodologies, such as weighted sum or lexicographic approaches [33], [34], under the assumption that decision-makers can precisely specify their preferences among different and often conflicting goals beforehand. However, in practice, such preferences may not be known in advance or may evolve over time, limiting the practical effectiveness of these traditional approaches. The increasing use of AI techniques has also led to different and potentially very promising approaches [35], including self-learning ones. Among those, RL is the one that has definitely accumulated the most attention in the research community [36]. RL is a promising ML area that has gained popularity in a wide range of research fields such as Network Slicing [37], Network Function Virtualization [38], and resource allocation [39]. More specifically, Deep Reinforcement Learning (DRL) has recently emerged as a compelling technique increasingly proposed in service management research [40]. DRL approaches aim to train a highly intelligent orchestrator to be able to make effective decision making in service management under various conditions. CI solutions represent another promising approach, utilizing smart and gradient-free optimization techniques that can efficiently explore a relatively large solution space [41]. Leveraging CI, it is possible to realize orchestrators that can explore relatively quickly even large solution spaces, thus being able to effectively operate in dynamic conditions with a reactive posture with a minimal re-evaluation lag. Advanced CI solutions seem to be well suited for expensive [42] and dynamic optimization problems [43], [44]. Although some metaheuristic performance analyzes in fog environments have been performed in recent times [45], establishing which of these solutions represents the most suitable one for service management in the CC is still an open research question.

RL and CI techniques have been proposed by the research community to address microservice deployment and resource management challenges at various layers of the CC. The firsts have shown promising results due to their ability to learn optimal strategies through continuous interactions with the environment [46]–[48]. Alternatively, metaheuristics offer an advantage in solving complex optimization problems more efficiently, often reducing computational time compared to traditional mathematical optimization techniques [49], [50]. The application of these methodologies has been further facilitated by the introduction of the DT paradigm also for CC ecosystems [51]. The DT concept has emerged in industry 4.0 as a high-accuracy virtual representation and software companion for physical assets related to applications ranging from maintenance [52] to process and network optimization [53]–[55]. However, more recently, the term has also been applied to software platforms and digital assets. Along that way, some

authors have already proposed DTs for mimicking and configuring/interacting with either networks, applications, or both [56]. In particular, DTs can address a wide range of situations by leveraging what-if scenario analysis and employing different mechanisms, ranging from performance optimization to chaos engineering [57]–[59]. Within this research avenue, a relatively recent development is the consideration of large-scale applications. As mentioned previously, large-scale applications are deployed on complex hybrid Cloud and CC scenarios, with many different public and private cloud environments and dynamic workloads, which present several challenges from the perspective of identifying optimal deployment configurations. In practice, it is very hard to determine the optimal configuration of computational and network resources for a large-scale application before their actual deployment. This task is even more complicated by the adoption of sophisticated orchestration platforms, such as K8s, that has become the de-facto solution for service management and orchestration, and it is increasingly proposed as a platform for a wide range of applications: from Network Function Virtualization (NFV) implementation [60] to the tactical edge domain [61]. Although this presents several advantages from the service provider perspective, it makes it even more difficult to assess upfront the proper configuration of a large-scale deployment, because its accurate evaluation must consider aspects that go well beyond the provisioning of resources, such as the number of Virtual Machines (VMs) to rent, their prices, etc., and needs to evaluate the runtime impact of K8s control loops on the application behavior.

This thesis analyzes and introduces different orchestration based mainly on the usage of RL, CI, and DT principles. Firstly, several simulations have been conducted to train, test, and compare RL and CI techniques in a realistic CC scenario characterized by limited computing resources at the edge and fog layers and unlimited resources at the cloud, considering metrics such as Percentage of Satisfied Requests (PSR) and Value-of-Information (VoI) through the Value-of-Information for Compute Continuum Ecosystems (VOICE) platform. In particular, VOICE supports dynamic service components allocation throughout the CC. A comparison between CI approaches and more traditional strategies (e.g. maximization of satisfied requests or latency minimization) evaluated the choice of the VoI-based service and resource management as the foundation of the VOICE framework.

The potential that RL showed in the first experiments was key to consider it as an approach to tackle the orchestration challenge in a multi-cluster infrastructure by proposing an RL-based Global Topology Manager (GTM) for efficient application deployment in K8s. An RL environment has been developed to provide a scalable and cost-effective solution to train RL agents for the multi-cluster orchestration problem. In addition, this thesis leverages the capabilities of the open-source project Kubernetes Armada (Karmada) [62], which acts as a control-plane solution for managing multi-cluster applications across hybrid cloud settings. The study aims to make Karmada's behavior more adaptive and intelligent than its current one by developing new components and novel orchestration policies to accomplish more efficient multi-cluster

scheduling. Motivated by the preliminary but promising results, this thesis proposes also a novel RL-based OpenAI Gym environment named *HephaestusForge* to provide a scalable and cost-effective solution to train RL agents for this problem. Instead of considering a single-objective reward model, HephaestusForge introduces a multi-objective reward function that considers three different performance factors: latency, cost, and inequality. This approach enables the learning of more effective deployment strategies capable of dealing with the various challenges that characterize CC scenarios. The results based on the OpenAI Gym environment show that RL can find efficient microservice placement schemes while prioritizing latency reduction, favoring low deployment costs, and avoiding distribution inequality compared to heuristic-based methods.

Despite the results obtained and the potential shown by the DS architecture, the complex and unpredictable interactions between heterogeneous objective functions in the CC make it challenging, if not impossible, to define precise and explicit preferences in advance. In practice, when dealing with multiple competing objectives, it is desirable to present to the decision-maker a set of multiple "optimal" trade-offs, such that well-informed decisions can be made a posteriori [63], [64]. In this regard, in contrast to single-objective optimization, a part of the work in this thesis aims to derive the entire Pareto Front (PF) of solutions, each achieving "optimal" trade-offs between three competing objectives: service latency, service deployment costs, and frequency of service interruption. Specifically, this thesis improves on the state-of-the-art by jointly considering the following four main aspects: *i)* simultaneous optimization of both scheduling and resource allocation, *ii)* assumption that the decision-maker's preferences between objectives are unknown before optimizing, *iii)* different pricing and latency models between heterogeneous clusters, profiled from real-world Amazon Web Services (AWS) instances, and *iv)* consideration of the possibility of splitting multiple microservices over different compute instances during resource allocation. In this context, a scheduling and resource allocation in K8s clusters has been formulated as a MO-Integer Linear Programming (ILP) problem and leverage metaheuristics to solve it efficiently.

The second part of this thesis claims the need to design novel DT solutions purposely implemented by considering K8s and the requirements of K8s-based applications. Those solutions would allow us to accurately capture the state of an existing K8s-based IT application through a virtual object with a smaller footprint and be capable of running what-if scenario analysis much faster than on a physical testbed. This would allow for an efficient and parallel evaluation of the behavior of the DT using different configuration parameters or even modified components, with many relevant applications ranging from design feedback to resource optimization and chaos engineering, as demonstrated by the implemented TELKA framework. The main contribution related to this aspect lies in KT, a comprehensive framework designed to implement DTs of K8s-based deployments. KT emulates the K8s orchestration functions and networking behaviors to be used as a guideline for service providers to accurately evaluate the impact of complex and large-scale K8s deployment scenarios. It allows the definition of

applications through a declarative description, which is semantically equivalent to K8s, to reenact and assess them through both generic and application-specific (and user-defined) Key Performance Indicators. KT enables its adopters to leverage the DT to identify optimized deployment configurations and evaluate different scheduling policies in complex computing scenarios where resources can be distributed at many levels, such as edge servers and cloud data centers. The contributions are manifolds: a description of the KT framework, a comprehensive presentation of a use case that contains not only the description of a container-based application that can be used as a reference for testing, but also a detailed CC scenario composed of edge and cloud resources. Among these, two solutions will also be proposed to model the statistical distribution of the response time of K8s applications. The first relies on a simulation-based inference approach, the second will be based on MDNs, which combine the use of conventional Feedforward Neural Network (FFN) with characterization by means of mixture models.

The final section of this thesis is focused on the usage of KT along with CE methodologies to improve the resilience of container-based applications deployed atop K8s, discussing the TELKA framework. TELKA can retrieve metrics, monitor information, and intervene in the event of performance degradation due to computing and network faults by adopting KT to learn how to mitigate adverse events of faults such as node failure or increased communication latency. KT enables to run multiple what-if scenario analyses during which the application is affected by injected faults. To deal with these faults, TELKA interacts with KT to learn a policy that can improve the resilience of the entire application. In this way, instead of interacting with the physical K8s application, TELKA learns by interacting with a DT, thus reducing the learning time and the operation costs related to the application of CE.

Chapter 2

Compute Continuum Foundations

Over the last two decades, the computing paradigm has undergone a severe evolution, guided by both technological advances and the emerging needs required by users and modern applications [65]. The emergence of Cloud Computing in the mid-2000s represented an historic model shift, offering on-demand provisioning of computational resources, scalability, and cost efficiency through a pay-as-you-go model. Cloud infrastructures quickly became a recognized de facto standard for data storage, large-scale analytics, and application components hosting [66], allowing organizations and adopters to focus more on software development rather than physical infrastructure management.

In parallel, the evolution of electronic devices has radically revolutionized the edge of the network. The proliferation of sensors, actuators, and the increasing computational power of mobile and embedded systems have led to IoT, one of the biggest recent innovations for academia, society, and industry [67]. Nowadays, billions of heterogeneous devices generate and consume massive amounts of data, enabling applications in domains such as smart cities [68], industrial automation [69], healthcare [70], and autonomous mobility [71].

This evolution, however, has soon exposed the limitations of Cloud-centric models. Although they can provide remarkable computation and storage along with additional support services, they often struggle in scenarios that require low latency, efficient resource utilization, data privacy, and security [5]. For instance, modern applications such as real-time video processing, augmented reality, and safety-critical systems cannot afford excessive delays derived from data transfers and processing in remote Data Centers (DCs).

To bridge this gap, intermediate layers have emerged between the Cloud and the IoT, giving rise to Fog and Edge Computing paradigms. These paradigms extend Cloud capabilities closer to the data sources, enabling distributed processing, hierarchical storage, and context-aware services [72]–[74]. Together, these layers form what is increasingly referred to as CC, a seamless and dynamic integration of Cloud, Fog, and Edge,

resulting in a distributed computing environment composed of heterogeneous devices as large-scale DCs down to resource-constrained embedded devices [75].

This chapter delves into the historical evolution of the enabling principles of the CC. It discusses how the continuum was born from the convergence of Cloud and IoT ecosystems, motivated by the limitations of centralized models, and how orchestration in such environments requires novel approaches leveraging AI techniques.

2.1 From Time-Sharing Systems to Cloud Computing

The concept behind Cloud Computing can be traced back in the past decades, when in the 1960s organizations like IBM and MIT started to conceptualize the notion of time-sharing systems. By implementing it, the main idea was to allow multiple users to simultaneously access a single mainframe, each interacting as if they had dedicated access to the machine. This would have led to an optimized usage of expensive physical assets and to an evolution beyond the optimization of mainframe computing capabilities. In 1962, John Mccarthy in "Time-Sharing Computing Systems" started to propose a first architecture of time-sharing systems, stating that *"Such a system will look to each user like a large private computer. The new applications that time sharing will make possible will be of as much additional benefit to science and management as resulted from the introduction of the stored-digital computer"* [76]. In his vision, each user was connected to a large and performing machine through a wired channel, with different consoles depending on the price. When a user required a service, he would have started writing a message request, with a queuing management to accommodate multiple requests simultaneously. His conceptualization included Round-Robin mechanisms to handle several programs at the same time and priority assignments to quantify the amount of time reserved to each program.

The next major step came with the introduction of the VM and its usage. In the 1970s, IBM introduced the Virtual Machine Facility/370 (VM/370) system, which allowed a single physical machine to be partitioned into multiple logical environments, each running its own operating system. To do so, it relied on three different operating systems: the Control Program to simulate multiple copies of the machine, the Conversational Monitor System system to support the personal use of a dedicated computer through a file system mechanism, and the Remote Spooling and Communications Subsystem to support data file transfers among computers and remote work stations linked properly [77]. Virtualization provided both efficiency and isolation, with resources that could be dynamically allocated while workloads remained secure and independent.

As computing systems became more distributed, the ability to securely connect to them remotely gained importance. The development and deployment of Virtual Private Networks (VPNs) during the 1980s and 1990s enabled organizations to extend their internal networks across public infrastructures, providing authenticated and encrypted

channels for communication. Their employment was mainly motivated by the growing trends related to user mobility and home offices, with companies providing reliable IT infrastructures to allow employees to access information and services from remote locations [78]. Although not Cloud systems in themselves, VPNs introduced the principle of virtualized access to resources beyond physical boundaries, a crucial step toward the geographically distributed nature of cloud platforms.

The real breakthrough toward modern Cloud Computing came in the early 2000s, thanks to significant advances in distributed computing, virtualization, and broadband connectivity. In 2006, Amazon launched the Elastic Compute Cloud (EC2) [79] and Simple Storage Service (S3) [80], which are widely considered between the first large-scale public Cloud services. Their pay-as-you-go pricing model and elastic scalability made it possible for companies of all sizes to access computational and storage resources without the need for large investments in hardware. This new paradigm based on "utility computing" introduces a remarkable shift: computing resources can now be requested and consumed as on-demand services. In the same year, Google CEO Eric Schmidt introduced the term to an industry conference: *"I don't think people have really understood how big this opportunity really is. It starts with the premise that the data services and architecture should be on servers. We call it cloud computing—they should be in a "cloud" somewhere"*. However, the debate about the origins of the term is still open and surrounded by many different theories [81]. Over time, the Cloud expanded beyond different types of service depending on users needs, from raw compute power to fully managed application delivery: Infrastructure-as-a-Service (IaaS) toward Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) [82].

In the current landscape, Cloud Computing has become the dominant paradigm for IT service delivery. This evolution has been further accelerated by the increasing adoption of containerization technologies such as Docker [83] and orchestration platforms like K8s [84]. These methodologies enabled an easier development of cloud-native applications specifically designed to exploit the scalability, resilience, and flexibility of the Cloud. Moreover, hybrid cloud [85] and multi-cloud strategies [86] became very popular, allowing organizations to distribute workloads across multiple providers while maintaining control over sensitive data.

2.2 The Evolution of IoT

The concept of the IoT that we know nowadays finds his roots into a series of experiments and innovations that gradually expanded the idea of connecting everyday objects to the Internet. One of the first known IoT devices comes from the Carnegie Mellon University, where a soda machine was modified to report its inventory status and the temperature of newly loaded drinks, allowing remote users to check availability before making a physical trip. To extract data from the machine, a board was installed to detect

the status of several indicator lights and connected to a gateway to the main computer of the department through ARPANET [87]. Building on similar ideas, John Romkey demonstrated that the Simple Network Management Protocol (SNMP) could be used to control physical devices, a toaster in that specific case [88]. Combining a novel SNMP application with a relay controlled directly by the parallel port on a portable computer, he was able to successfully control the power and toast calibration of the device. Around the same time, in 1991, researchers at the University of Cambridge developed what became the world's first webcam to monitor the Trojan Room coffee pot. The system was mainly created to prevent wasted trips to an empty coffee pot, streaming low-resolution images over the local network [89]. To do so, a fixed camera was linked to a video framegrabber through a wired connection. Thanks to a client application that everyone could run, it was possible to obtain an icon-sized image of the coffee pot, which was updated around every 20 seconds.

Parallel to these developments, the foundations of modern IoT were reinforced by progress in global infrastructure. In 1995, the Full Operational Capability of the first version of the long-running Global Positioning System (GPS) satellite program was declared by the U.S. government. It provided two different levels of service: a Standard Positioning Service available to all users worldwide with a 95% of precision, and a highly accurate military positioning, velocity and timing service available to users authorized by the U.S called Precise Positioning Service [90]. Other than infrastructures, additional technical advancements also contributed to the evolution of IoT. In 1998, IPv6 was standardized by the Internet Engineering Task Force (IETF), providing an expanded address space essential to support the vast number of devices envisioned in an IoT ecosystem [91]. In fact, while 32-bit IPv4 only provides enough unique identifiers for around 4.3 billion devices, 128-bit IPv6 has enough unique identifiers for up to 2^{128} different devices.

In 1999, the expression was first introduced by Kevin Ashton to describe a system in which the Internet is connected to the physical world via ubiquitous sensors and Radio Frequency Identification (RFID) technology [92]. His vision leveraged the potential of tagging and tracking objects in a unique way through RFID, enabling a more automated and accurate representation of the real world digitally and, at the same time, reducing human dependency. Throughout the early 2000s, the proliferation and practical realization of IoT was fueled by the maturation of several technologies and the development of others. For instance, the decreasing cost and miniaturization of sensors, wireless modules, and embedded processors made it feasible to integrate connectivity into a growing variety of devices. Furthermore, the launch of the first iPhone in 2007 and Android-based smartphones shortly after introduced a completely new and novel way for general users to interact with the world and devices connected to the Internet [93]. In fact, Smartphones not only expanded connectivity, but also introduced key enablers such as integrated GPS, accelerometers, and later, Near Field Communication (NFC), which served as prototypes for many IoT applications.

Another significant milestone in IoT research and development came in 2008, when the first dedicated international conference was organized in Zurich, Switzerland [94]. This event represented a first signal of consolidation of IoT as a distinct field and consolidated technology subject of study and innovation from both industry and academia. In fact, this was the first year that the number of estimated IoT devices grew to surpass the number of humans on earth and scientific publications showed a first impactful growing trend, along with several action plans related to IoT issued by many nations [95].

From the early 2010s onward, the IoT has progressively evolved from experimental deployments into a global technological paradigm with a profound impact across multiple sectors. Several innovations and novel integrations with other paradigms characterized this phase, from smart home markets to advances in automotive technologies. From a consumer-oriented perspective, the launch of Amazon Echo in 2014 significantly paved the way for the personal assistant market. Lately, these foundations have become crucial for an integration with business and corporations, such as Alexa for Business [96]. In parallel, AI and ML enhanced the ability to extract valuable insights from massive streams of sensor data, enabling predictive maintenance [97], adaptive automation, and personalized user experiences [98]. Blockchain has also been explored as a means to improve security, traceability, and decentralized trust management in IoT networks [99].

The reliance on IoT for sensor integration, Vehicle-to-Vehicle (V2V) [100] and Vehicle-to-Infrastructure (V2I) communication [101], and real-time decision-making are enabling the rise and consolidation of connected and autonomous vehicles. This innovation has been further accelerated by the combination of IoT and cloud services, allowing the deployment of smart city ecosystems that leverage services ranging from intelligent traffic management to environmental monitoring [102] and energy optimization [103].

As mentioned previously, due to scalability and latency requirements, edge and fog computing emerged as main solutions, bringing computational resources closer to the data sources rather than relying solely on centralized cloud infrastructures. By seamlessly spanning cloud, fog, and edge-level resources, IoT infrastructures call for a continuum of computational capabilities capable of dynamically allocating processing power, storage, and networking functions where they are most effective. In fact, these resource-constraint devices can delegate to powerful cloud services the execution of computationally intensive tasks and to dedicated hardware close to end-users a limited latency and resource-saving task execution required in a wide range of modern applications.

2.3 Beyond Cloud and IoT Boundaries: The Compute Continuum

The historic evolution of IoT resulted in the generation and processing of huge data sets at the edge of the network, making the cloud alone unsuitable to properly deal with such heterogeneity of data and application requirements. Therefore, edge and fog computing quickly become effective solutions to overcome cloud limitations, improving aspects like average response time and bandwidth usage. With the continuous evolution of the global IT infrastructure and the need to connect devices in a distributed network to meet a very fragmented range of requirements, the CC takes shape [104]. This paradigm enables users to rely not only on the resources available at the edge, but also to leverage the computing capabilities offered by fog and cloud infrastructures. As a result, the CC introduces new opportunities for distributing the workload of computationally intensive services, including machine learning applications, online gaming, and large-scale data management [2], [3].

2.3.1 Definition and Architecture Discrepancies

Since the research interest that the CC has drawn from many academics and adopters in the last few years and the lack of universal standards, the current literature presents a very heterogeneous profile, with several different terminologies, perspectives, architectures, and conceptual overlaps [1]. From a terminology perspective, the most used ones are typically *"Cloud Continuum"*, *"Edge-to-Cloud Continuum"*, and *"Compute Continuum"*, with the first and the latter often used interchangeably. However, *"Cloud Continuum"* seems to emphasize more the cloud layer of the continuum, putting less importance to edge and fog layers, which could be crucial, especially in modern applications that have very strict requirements regarding some metrics such as latency. Therefore, for this thesis, the terminology *"Compute Continuum"* has been considered the most proper and elegant one, highlighting more the computing capabilities of the ecosystem rather than specific locations or computing facilities.

In parallel, the definitions and architectures proposed by various organizations have also been subject to numerous significant changes. For a long time, the most common vision of the CC referred to the seamless integration of heterogeneous computing resources distributed across the edge, fog, and cloud layers (as illustrated in Figure 2.1). At the Edge layer, computational resources are primarily provided by Multi-Access Edge Computing (MEC) servers deployed at Base Stations (BSs). These servers connect end users to both the MEC infrastructure and the core network, offering the closest computational proximity to end devices. Owing to the very low communication latency they can achieve (typically in the range of 1–10 ms), MEC servers are well suited for hosting latency-sensitive and mission-critical applications.

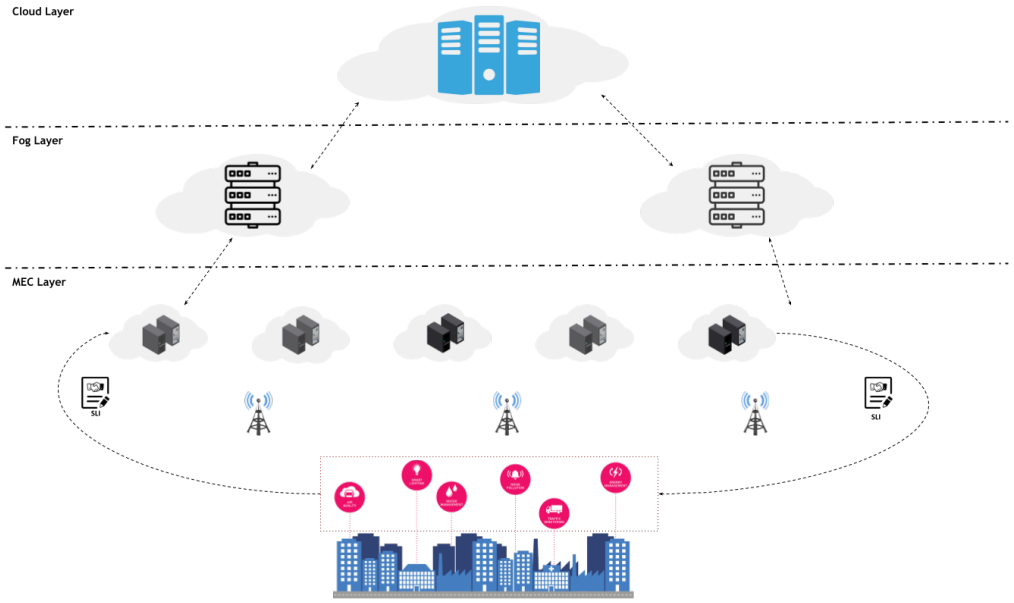


Figure 2.1: A CC architecture proposal shows computing resources deployed across the edge, fog, and cloud layers.

Above the edge, the Fog layer offers enhanced computational capacity compared to MEC servers, albeit with moderately higher communication latency. This intermediate layer serves as a trade-off between performance and responsiveness, enabling a wider set of applications to be executed closer to end users without fully relying on the remote cloud. At the highest level of the continuum, the cloud layer provides virtually unlimited computational and storage resources. These resources are best suited for large-scale batch processing tasks and applications with relaxed latency constraints.

However, stakeholder requirements have evolved throughout the years, especially from realities such as industries and governments, where there is a growing need to maintain control over edge and cloud technologies and to set up on-premise facilities to guarantee privacy and low latency. Therefore, to address them, several organizations have proposed several alternative architectures, including the European Commission [105]. As demonstrated in Figure 2.2, this vision has a more cloud-centric perspective than user-centric and the layers are classified according to their distance from the cloud facilities, which are considered the main computational centers for large-scale data processing and storage:

- Similarly to other proposals, the computational capacity and main services such as security and monitoring are offered by the major **cloud services and telcom providers**, such as AWS or Microsoft Azure. These DCs are publicly shared in a multi-tenant way and provide hardware and software to tackle a wide range of

different use cases.

- The **Near Edge** layer is composed mainly by micro-DCs and regional cloud nodes within hundreds of kilometers far from the devices. The characteristics are comparable with those of the previous layer: multi-tenancy mode, public, and with hardware and software resources to address a large number of different use cases according to users needs.

- The **Far Edge** covers an area that is included in tens of kilometers far from the devices, usually between 50 and 100 kilometers. In this layer the computing resources are closer to end-users and are reserved mainly to perform faster data aggregation and pre-processing tasks. These resources are gathered in more small-scale DCs which are deployed near to strategic hotspots (e.g., shopping malls and industrial facilities). The multi-tenancy still stands, but hardware and software equipment begins to be more specialized to handle particular types of task.

- The **On-Premise Edge** layer comprehends computational resources very close to data sources (e.g., street lights or BSs), typically not more than 5 kilometers. These nodes can be located in strategic places such as stadiums or industrial plants, and are meant to be completely private. Their scope depends on specific needs of their adopters and they eventually may require additional features, such as real-time connectivity.

- The **On-device** is the closest layer to end users and is composed almost entirely by IoT devices that perform local data collection and processing, enabling real-time computation and decision making, which is crucial in a wide range of modern scenarios.

Since the research presented in this thesis is focused more on academic use cases rather than industrial application scenarios, the architecture shown in Figure 2.1 is considered the baseline.

2.3.2 Resource and Service Management in the Compute Continuum and Other Major Challenges

Realizing the full potential of the CC remains an ambitious goal due to a number of multifaceted challenges that span technical, organizational, and even economic dimensions. At the federated or multi-cluster level, one of the most critical problems concerns system-level orchestration and inter-cluster coordination. Although modern cloud-native infrastructures already rely on mature scheduling and autoscaling mechanisms within individual clusters, thanks to frameworks such as K8s or OpenShift, these solutions are largely limited to single administrative domains. Extending such capabilities to

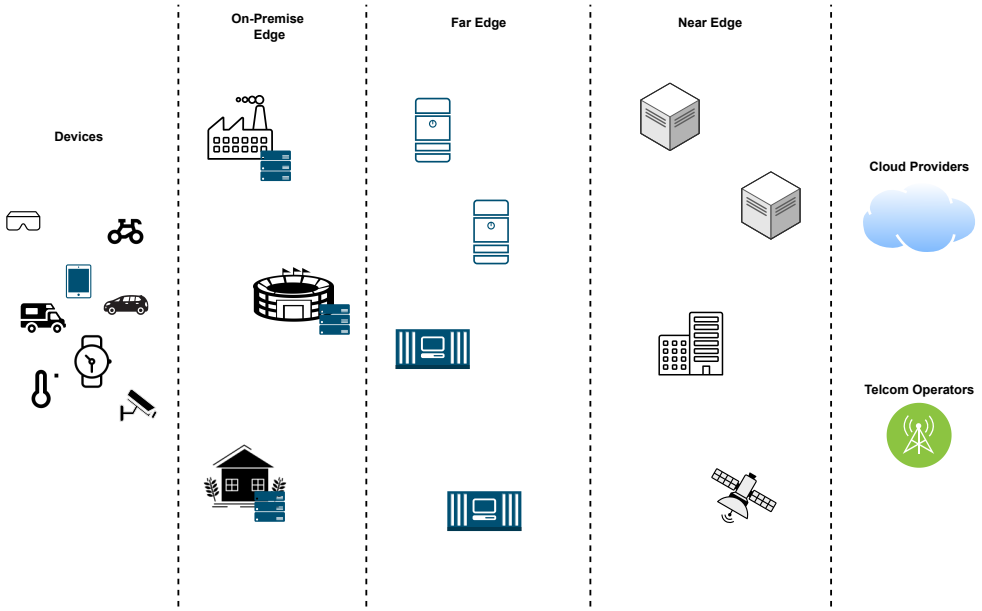


Figure 2.2: A CC architecture based on a cloud-centric perspective to address emerging requirements more related to industry and governments.

large-scale federations of clusters distributed across heterogeneous administrative, geographic, and technological boundaries remains an open research problem. In particular, reliable and automated approaches for global scheduling, dynamic scaling, and service migration across cluster federations have not yet reached production-level maturity. The main difficulty lies in achieving efficient and stable distributed decision-making. Without appropriate coordination and control mechanisms, autonomous cluster-level agents can enter conflicting feedback loops, often referred to as "ping-pong" effects, where competing management entities continuously override previous decisions. This may lead, for instance, to inefficient task migrations or oscillatory load redistribution that drop performance rather than optimize it.

Beyond orchestration, realizing the promise of a continuous computing environment also requires addressing resource reorganization and resilience in the face of network discontinuities. Connectivity interruptions require mechanisms that go beyond classical disconnection-resilience patterns, such as circuit breakers in microservice architectures, but more proactive approaches that involve topology control strategies, whereby clusters can be either activated or deactivated dynamically, or even physically relocated to respond to evolving conditions or requirements. Such mechanisms become particularly relevant in mission-critical scenarios, where continuity of service and locality of computation are predominant.

At the economic and governance level, another open question concerns the busi-

ness models that will sustain resource sharing across the CC. On one hand, public continuum providers could emerge that, analogously to cloud providers, can deploy and manage small-scale data centers at the edge to offer resources as a service. On the other hand, community-driven or cooperative models may prevail, in which individuals, enterprises, and institutions federate their own computational resources and allow adopters to rent or bid for currently available capacity. Hybrid approaches are also a possible paradigm, combining commercial offerings with decentralized and community-based proposals. Regardless of the model that proves to be superior, the design of orchestration and management frameworks must be closely aligned with the economic and trust assumptions that serve as basis of these ecosystems.

From a service management perspective, the CC involves multiple stakeholders. Service providers may request the deployment of one or more MEC applications on the available resources. To this end, they rely on an infrastructure provider, which is responsible for provisioning and managing applications throughout the continuum. The infrastructure provider must therefore accommodate a variety of provisioning requests using a distributed and heterogeneous pool of computing resources. This requires determining suitable allocations that maximize the number of services deployed while also optimizing their performance under current operating conditions, such as fluctuating resource availability and device heterogeneity. Within a distributed and heterogeneous environment such as the CC, the ability to effectively manage services becomes essential to ensure seamless delivery and optimized use of resources. At the same time, coordinating resource allocation, utilization, and management across the different layers of the CC remains a significant challenge.

The allocation problem is further complicated by the diverse characteristics of the computing layers. For example, mission-critical services often impose strict latency requirements that can only be met by edge resources, which are inherently constrained in capacity. In contrast, while the cloud can provide abundant resources, its higher latency makes it unsuitable for such use cases. Balancing these competing requirements highlights the fundamental challenge of resource management in the CC. Additionally, the introduction of innovative options and the integration of a wide range of devices into the ecosystem also raise a considerable number of potential threats, which in turn demand the adoption of effective countermeasures [4]. Addressing these complexities requires management solutions capable of allocating multiple service components over a limited and distributed set of devices, while also accounting for the specific characteristics of the resources available at each layer. Furthermore, the high level of dynamicity inherent in CC scenarios highlights the need for adaptive and intelligent orchestration mechanisms that can autonomously learn how to identify and apply the most suitable service allocations [106]. To this end, several models and frameworks have been developed throughout the years, targeting many different orchestration objectives such as resource savings, availability, and scalability [107], [108].

In conclusion, it is important to note that resource sharing at scale inevitably raises

cybersecurity concerns. The shift from isolated, single-tenant environments toward shared, multi-tenant distributed infrastructures significantly enlarges the attack surface. While such environments bring clear benefits in terms of cost efficiency and reduced time-to-market, they also introduce new threat models and trust management challenges [109]. Although not a topic of this research thesis, ensuring security across the continuum will require revisiting existing protection mechanisms, strengthening authentication, authorization, and isolation policies, and possibly developing continuum-aware security frameworks that account for the heterogeneity and dynamism of the environment.

2.4 Chapter Summary

This chapter discussed in-depth the origins of the CC, illustrating the enabling technologies and innovations that led to the birth of this paradigm to overcome limitations and favor the integration of technologies with very different requirements and objectives.

Section 2.1 outlined the birth and evolution of the cloud computing paradigm. Starting from the conceptualization of time-sharing systems by John Mccharty, the development of the first VM and the deployment of VPNs revolutionized the access to remote resources and services and the management of IT infrastructures. Building on this principles, cloud computing has become the predominant technology for service delivery and remote computation very quickly, demonstrating the massive impact that this paradigm shift had in the community.

Section 2.2 explored the foundations and the proliferation of the IoT, analyzing the maturation of the connection between physical and digital worlds through the Internet thanks to the advancements of the global IT infrastructure and the ever growing need to integrate many different technologies to address user needs.

Finally, Section 2.3 detailed the main characteristics of the CC, describing the current state-of-the-art and misalignment related to terminology, definitions, and architectures. This section also discusses the main challenges related to this ecosystem and the main problems that need to be addressed, which call for innovative and smart orchestration solutions.

Chapter 3

Reinforcement Learning and Computational Intelligence Background

The increasing adoption of AI techniques has introduced a variety of promising approaches, particularly those that incorporate self-learning capabilities [35]. Among these, RL has attracted significant attention within the research community [36]. RL, as a branch of ML, has been successfully applied in various domains, including network slicing [37], NFV [38], resource allocation [39], and network congestion control [110]. More recently, DRL has emerged as a compelling technique, widely proposed for service management [40]. By extensively training an intelligent orchestrator, DRL enables effective decision-making for service allocation and management under a variety of operating conditions.

In parallel, CI solutions offer another promising research avenue. These methods rely on gradient-free optimization techniques that can efficiently explore large solution spaces, making them well suited for highly dynamic environments [41]. CI-based orchestrators can react quickly to changes, thus being able to effectively operate in dynamic conditions with a minimal re-evaluation response time. Advanced CI approaches have proven particularly effective in addressing complex and computationally expensive optimization problems, as well as in scenarios characterized by strong dynamicity [43], [44]. Although some metaheuristic performance analyzes in fog environments have been conducted in recent times [45], establishing which of these solutions represents the most suitable one for service management in the CC is still an open research question.

This chapter reviews extensively the technical principles of the RL and CI approaches leveraged in this thesis, highlighting advantages and disadvantages of each one. Furthermore, it introduces novel approaches implemented for this research based on RL fundamentals, such as DS-RL. Finally, it explores in depth the MO variations of both

approaches, highlighting their peculiar ability to produce several possible trade-offs between different conflicting objectives to enable informed a posteriori decisions.

3.1 Computational Intelligence

Within the resource-management field, CI methods have accomplished remarkable results thanks to their capability of efficiently exploring large and complex spaces. In particular, they can generate feasible and robust solutions that often prove more advantageous than traditional approaches [111], which, although effective in specific scenarios, tend to face limitations such as scalability issues. CI methods are based on gradient-free (or black-box) optimization, relying on the evaluation of a relatively large number of samples across significant regions of the search space to approximate or identify the global optimum. Due to the typically considerable costs of repeatedly involving the actual infrastructure, applying these methods directly to a real system is often considered unsustainable. Therefore, they usually require the use of a representative system model that can act as the objective or target function, as represented in Figure 3.1. In this way, the CI algorithm operates on the model to identify optimal solutions that can subsequently be applied to the real system.

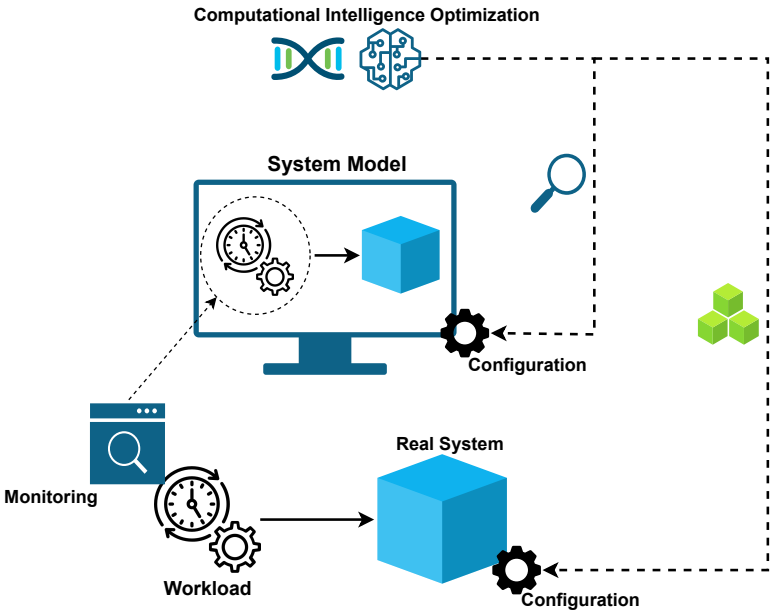


Figure 3.1: CI interaction with a real system to perform optimization.

Essentially, CI approaches progressively build knowledge of the target function by storing the results of their sampling in a memory pool, which evolves over time to guide

the search toward the most promising regions of the solution space. These methods often draw inspiration from evolutionary and biological processes (in the literature they are often indicated as '*Evolutionary Computation*' methodologies or '*Metaheuristics*'), enabling efficient exploration and exploitation. Despite the proven high effectiveness for the optimization of static systems, several challenges arise when applied to scenarios characterized by strong dynamicity, as in the CC case [57]. In particular, the use of CI for dynamic and computationally expensive optimization problems raises the question whether to invalidate or discard previously accumulated knowledge and to what extent. Addressing this issue remains an open research problem and is currently the focus of growing attention in the scientific community [112], including this thesis. Since there are numerous CI solutions in the current literature, this section illustrates and motivates the selection of the metaheuristics exploited in this research.

3.1.1 Genetic Algorithms

The Genetic Algorithm (GA) is a metaheuristic inspired by the principles of biological evolution. It operates on a population of individuals, each representing a candidate solution to the optimization problem considered. An individual is typically described by a genotype, which encodes its position in the search space, and a phenotype, corresponding to the evaluation of the objective function at that position. The phenotype represents a "fitness" value, which reflects how well the solution adapts to the given environment. As illustrated in Algorithm 1, to discover the potential optimum of the optimization problem, the GA evolves its population across multiple generations through evolutionary operators such as selection, recombination, and mutation, which generate new individuals with better fitness values. Selection mechanisms favor individuals with higher fitness, while recombination and mutation introduce diversity by exchanging or altering the characteristics of each potential solution found. This process allows the algorithm to guide the solutions toward promising regions across generations while maintaining enough variability to avoid premature convergence, making this methodology capable of exploring the search space in a robust and efficient manner.

The representation of the genotype can vary depending on the problem, ranging from bit-strings to integers or real-valued encoding, and this choice has a significant impact on the convergence speed and quality of solutions [113]. Similarly, different selection, recombination, and mutation operators can be adopted, enabling a balance between exploration and exploitation. For example, binary tournament selection is widely used due to its simplicity, low computational cost, and resilience to excessively exploitative behavior, which other selection operators often exhibit [114]. Common recombination operators include one-point, two-point, and uniform crossover, while mutation can be implemented through techniques such as bit flips or geometrical distribution-based displacements [115].

By tuning the operators and parameters that govern selection and mutation, the

Algorithm 1 Genetic Algorithm.

```
1:  procedure GA(target_function, conf)
2:      P ← initialize_random_population(conf)
3:      generation ← 1
4:      fittest ← evaluate_population(P, target_function)
5:      repeat
6:          P_next ← ∅
7:          parents ← select_parents(P, conf)
8:          for all p_1, p_2 ∈ parents do
9:              c_1, c_2 ← crossover(p_1, p_2, conf)
10:             P_next ← mutate(c_1, conf), mutate(c_2, conf)
11:         end for
12:         P ← P_next
13:         for all m ∈ P do
14:             m.fitness ← target_function(m.genotype)
15:         end for
16:         fittest ← argmax_{m∈P} m.fitness
17:         generation ← generation + 1
18:     until generation ≥ conf.max_generations
19:     return fittest.genotype, fittest.fitness
20: end procedure
```

configuration of the GA can be modulated to suit specific problem domains, enhancing either explorative or exploitative behaviors depending on specific needs. Ultimately, the goal of a GA is to evolve towards an optimal population. In other words, a situation in which further generations produce offspring that no longer differ significantly from their predecessors.

3.1.2 Particle Swarm Optimization and Quantum Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a metaheuristic that models the collective behavior of a swarm of particles that navigates the search space of an optimization problem looking for the potential global optimum. Each particle represents a candidate solution, and its position is evaluated through a defined objective function. The overall process, often referred to as *Swarm Intelligence*, relies on the exchange of information among particles to guide the search for promising regions of the solution space. The way in which this information is aggregated and shared is central to improving the efficiency and effectiveness of the optimization process [116].

PSO has attracted considerable attention over the years due to its simplicity of im-

plementation and the relatively small number of parameters that must be tuned to achieve a suitable balance between exploration and exploitation. However, identifying optimal parameter values remains a non-trivial task, and significant research has been conducted to refine the algorithm and achieve its state-of-the-art [117], [118]. Fundamentally, PSO is inspired by the social dynamics of bird flocks, where a population of interacting particles collaboratively explores the search space [119].

At each iteration t, each particle i has:

- A current **position vector** $X_i(t) = (X_{i,1}(t), \ldots, X_{i,N}(t))$, whose components represent the decision variables of the problem;
- A **velocity vector** $V_i(t) = (V_{i,1}(t), \ldots, V_{i,N}(t))$, which captures the movement of the particles;
- A **particle attractor** $P_i(t) = (P_{i,1}(t), \ldots, P_{i,N}(t))$, representing the best position that the particle has encountered so far related to the considered objective function.

Therefore, the movement of particles is described by the following equations:

$$\begin{aligned} V_{i,j}(t+1) &= V_{i,j}(t) + C_1 * r_{i,j}(t) * (P_{i,j}(t) - X_{i,j}(t)) + C_2 * R_{i,j}(t) * (G_j(t) - X_{i,j}(t)) \\ X_{i,j}(t+1) &= X_{i,j}(t) + V_{i,j}(t+1) \end{aligned} \tag{3.1}$$

in which $G(t)$ is the swarm attractor, representing the best position the whole swarm has encountered so far, $r_{i,j}(t)$ and $R_{i,j}(t)$ are random sequences uniformly sampled in $(0, 1)$, and C_1 and C_2 are constants.

An important evolution of PSO is represented by Quantum Particle Swarm Optimization (QPSO). Unlike classical PSO, where particles follow Newtonian random walks, QPSO introduces a quantum behavior that governs particle movement. This significantly simplifies the configuration process, reducing the number of required parameters to a single contraction–expansion coefficient (α) [120]. This parameter effectively acts as a tuning "knob", regulating the trade-off between local and global search. QPSO has been shown to address some of the limitations of classical PSO, particularly its tendency to become trapped in local minima, and has demonstrated superior performance in a variety of benchmark problems [121].

Algorithm 2 shows the steps necessary to implement the QPSO.

Algorithm 2 Quantum-inspired Particle Swarm Optimization.

procedure QPSO(target_function, conf)
    $particles \leftarrow initialize\_random\_swarm(conf)$
    $iteration \leftarrow 1$
    $best \leftarrow null$
    repeat
        for all p in $particles$ **do**
            $p.val \leftarrow target\_function(p.pos)$
            if $p.val > p.bestval$ **then**
                $p.bestpos \leftarrow p.pos$
                $p.bestval \leftarrow p.val$
            end if
            if $p.val > swarm\_bestval$ **then**
                $swarm\_bestpos \leftarrow p.pos$
                $swarm\_bestval \leftarrow p.val$
            end if
        end for
        $mean\_bestpos \leftarrow \frac{1}{M} \sum_{i=1}^{M} particles[i].bestpos$
        $\vec{\phi} \leftarrow rand()$
        $\vec{u} \leftarrow rand()$
        $s \leftarrow rand()$
        $attr \leftarrow \vec{\phi} * p.bestpos + (1 - \vec{\phi}) * swarm\_bestpos$
        $\delta \leftarrow \alpha * |p.pos - mean\_bestpos| * \ln(1/\vec{u})$
        $p.pos \leftarrow attr + sign(rand() - 0.5) * \delta$
        $iteration \leftarrow iteration + 1$
    until $iteration > conf.max\_iterations$
    return $swarm\_bestpos, swarm\_bestval$
end procedure

3.1.3 Multi-Swarm Particle Swarm Optimization

Although PSO has been successfully applied to a wide range of applications, its effectiveness decreases in dynamic environments, where it suffers from outdated memory and lack of diversity [122]. To address these shortcomings, researchers have explored Multi-Swarm PSO (MPSO) approaches [123]. MPSO enhances diversification, thereby mitigating premature convergence, through two key mechanisms: the use of multiple swarms and the introduction of repulsion. By instantiating several swarms, the algorithm can simultaneously explore different regions of the search space, which is particularly beneficial in problems characterized by multiple optimal solutions (i.e., multi-peak target functions). Repulsion is applied both within individual swarms and across

different swarms, maintaining diversity throughout the optimization process.

A notable example of MPSO is the variant inspired by atom dynamics [122]. In this construction, each swarm is divided into a dense nucleus of neutral (or positively charged) particles and a surrounding nebula of negatively charged particles. The neutral portion evolves according to classical PSO definition (Equation 3.1), while negatively charged particles follow quantum-inspired dynamics. Some implementations also adopt Coulomb force-inspired repulsion, although quantum-inspired models have proven simpler and at least equally effective [123].

In addition to these mechanisms, MPSO maintains diversity through exclusion and anti-convergence strategies, resulting in a continuous birth-and-death process for swarms. Exclusion provides local diversity by preventing multiple swarms from converging to the same optimum: when two swarms S_A and S_B come closer than a predefined exclusion radius r_{excl}, one is terminated and replaced with a new randomly initialized swarm S_C. Anti-convergence, on the other hand, enforces global diversity by ensuring that at least one swarm continues patrolling the search space rather than converging. This is achieved by monitoring swarm diameters and replacing one swarm if all fall below a threshold dynamically estimated to suit the characteristics of the optimization problem considered.

Designed specifically for dynamic optimization problems, MPSO techniques currently represent state-of-the-art solutions in this area [43], and therefore constitute a particularly promising direction for orchestration challenges in the CC.

3.1.4 Grey Wolf Optimization

Grey Wolf Optimization (GWO) is another nature-inspired metaheuristic, modeled on the leadership hierarchy and cooperative hunting strategies of grey wolves [124]. The algorithm reproduces the social structure of a grey wolf herd and its coordinated behavior when searching and hunting for prey. The search agents correspond to wolves and are organized into four hierarchical groups: alpha, beta, delta, and omega, which collectively guide the movements of the pack during the optimization process.

From a mathematical perspective, the GWO algorithm begins by initializing a population of wolves, with each individual representing a candidate solution in the search space. The fitness of these solutions is then evaluated using a problem-specific objective function. The three best-performing solutions are selected as the leading wolves: the alpha corresponds to the best solution, followed by beta and delta as the second and third best, respectively. The remaining candidates are classified as omega wolves, which adapt their movements by following the leaders throughout the search process. Their position update equations are the following:

$$
\begin{aligned}
A^1 &= 2a \times \text{rand}() - a \\
C^1 &= 2 \times \text{rand}() \\
D^\alpha &= |C^1 \times \alpha - X_i| \\
X^1 &= \alpha - A^1 \times D^\alpha
\end{aligned}
\tag{3.2}
$$

Similar equations are used to calculate X^2 and X^3 based on the positions of the beta and delta wolves, respectively. The new position of each wolf is then updated as follows:

$$
X_i = (X^1 + X^2 + X^3)/3 \tag{3.3}
$$

Conceptually, the leading wolves in GWO define the boundaries of the search space, which can be represented as a circle in two dimensions or, more generally, as a hypercube or hypersphere in higher dimensions. The algorithm relies on two key components, denoted as A and C, that guide the wolves in their search for fitter prey and help prevent premature convergence to local optima. While A decreases linearly over time to balance exploration and exploitation, C operates differently, as it does not include a decreasing component. Instead, $C > 1$ emphasizes exploration, while $C > 1$ encourages exploitation, allowing the algorithm to dynamically adjust the behavior of the search agents throughout the process. The hunting procedure is repeated iteratively for a fixed number of iterations or until the convergence criteria are met, with the alpha wolf representing the best solution identified. Algorithm 3 represents the technical steps described above.

Algorithm 3 Grey-Wolf Optimization (GWO).

Initialize the grey wolf population X_i (i = 1, 2, …, n)
Initialize a, A, and C,
Compute the fitness of each agent
X_α = best solution found by a wolf
X_β = second-best solution found by a wolf
X_δ = third-best solution found by a wolf
while t < **Max number of iterations do**
    for each wolf X_i **do**
        Update the position of X_i by Equation (3.3)
    end for
    Update a, A, and C
    Compute the fitness of each agent
    Update X_α, X_β, *and* X_δ
    $t = t + 1$
end while
Return X_α

Similarly to PSO, GWO does not enforce direct relationships between search agents and the fitness function [124], allowing effective integration of penalty mechanisms into the model constraints. This characteristic makes GWO particularly suitable for tackling complex constrained optimization problems relevant to orchestration in the CC. Furthermore, this will be the only metaheuristic approach considered in this thesis other than GA and PSO. The reason behind this choice lies in the fact that a lot of CI methodologies present in the literature generally propose unclear and misleading novelties and mathematical models, which often result to be variations of PSO [125].

Table 3.1: Qualitative comparison of the considered metaheuristics

Characteristic	**GA**	**PSO**	**QPSO**	**MPSO**	**GWO**
Population-based	✓	✓	✓	✓	✓
Explicit diversity preservation	✓	×	×	✓	×
Low number of hyperparameters	×	✓	✓	×	✓
Adapted for dynamic environments	×	×	×	✓	×
Multi-population / multi-swarm	×	×	×	✓	×
Susceptible to premature convergence	×	✓	×	×	×
Constraint handling via penalties	✓	✓	✓	✓	✓
High exploration capability	✓	✓	✓	✓	✓

As summarized in Table 3.1, MPSO has the potential to be the most suitable metaheuristic candidate for dynamic CC scenarios, thanks to its explicit diversity preservation and multi-swarm structure, whereas QPSO offers an attractive trade-off between exploration capability and configuration simplicity.

3.2 Reinforcement Learning

RL adopts a significantly different paradigm compared to optimization methods such as CI. As depicted in Figure 3.2, it is based on a trial-and-error training process in which an agent learns to improve its behavior by interacting with an environment through sequences of actions, each followed by a reward signal that quantifies how much the decision made is beneficial to attaining the final goal. The optimization problem is typically conceived through a Markov Decision Process (MDP), which in the most common formulation is defined as a tuple $< S, A, P_a, R_a >$. More specifically:

- **S** represents the **state space**. In other words, the observations available to the agent at every timestep t.
- **A** is the **action space**, which represents the set of all possible actions that the agent can perform at each timestep t.

- $P_a(s, s')$ is the **state transition probability function**, which defines the probability that performing action a in the state s at time t will lead to state s' at time $t + 1$.
- $R_a(s, s')$ defines the **reward received** after transitioning from state s to state s' due to action a.

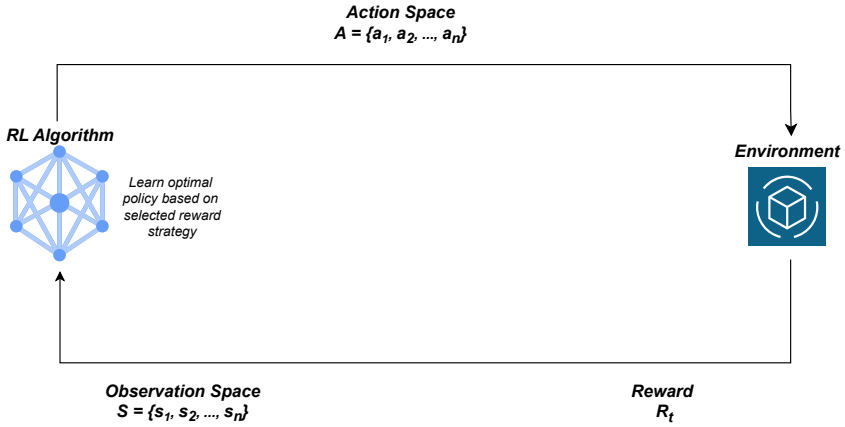


Figure 3.2: Overview of the RL architecture

Depending on the complexity of the problem, other formulation of the MDP are typically employed, such as the Partially MDP [126] or the Constrained MDP [127].

The primary goal of RL is to derive an optimal policy that guides the decisions of the agent in each state. Formally, this requires learning the set of parameters θ for a policy π_θ that maximizes the expected return:

$$\max_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] \tag{3.4}$$

The reward model therefore plays a central role in the performance of RL solutions, and its design remains one of the key challenges in this field. There is a continuous and active debate on how to define reward functions that are both effective and aligned with the desired system objectives [36], [128].

Although RL is not strictly an optimization technique, it provides a flexible framework that can be adapted to a wide range of scenarios. Unlike CI methods, which often face memory invalidation issues in dynamic environments, RL is inherently designed to deal with changing conditions. Policies learned through robust RL training can generalize across different situations, and when system dynamics evolve, policy re-evaluation in the updated state can identify suitable actions without discarding the accumulated knowledge base. Another important advantage that makes RL particularly appealing

for orchestration in the CC is that it can be applied directly to real systems, learning from them through continuous iterations [129].

3.2.1 From Q-Learning to Deep Q-Network

Q-Learning is one of the most widely known RL algorithms. It constructs a memory structure, typically known as the Q-Table, that stores the Q-values associated with each possible state–action pair. Each Q-value represents the expected return of executing a given action a_t at a specific time step t, following the policy from the next state onward. The update of Q-values is governed by the Bellman equation:

$$Q(S_t, A_t) = (1 - \alpha)Q(S_t, A_t) + \alpha \times (R_t + \lambda \times \max_a Q(S_{t+1}, a)) \tag{3.5}$$

where S_t and A_t denote the state and action at time t, R_t is the reward received, α is the learning rate, and λ is the discount factor, which is modeled to prioritize immediate rewards [130]. Through repeated interaction with the environment, the agent incrementally refines its estimates of future rewards and converges towards policies that maximize long-term return.

Although conceptually simple and effective, Q-Learning struggles when applied to problems with large or continuous state and action spaces, as maintaining and updating a Q-Table with hundreds or thousands of entries can quickly become infeasible in terms of both memory and computation. Deep Q-Learning overcomes the limitations of Q-Learning by approximating the Q-Table with a deep neural network, which optimizes memory usage and gives a solution to the curse-of-dimensionality problem, forming a more advanced agent called the DQN [131].

$$L(\theta) = L(R_t + \gamma \max_a Q(S_t, a),\ Q(S_t, A_t)) \tag{3.6}$$

$$\theta = \theta - \alpha \nabla L(\theta). \tag{3.7}$$

Given a state representation as input, the network outputs the estimated Q-values for all possible actions. The network parameters are optimized using variants of the Bellman equation, where the loss is computed between the predicted and target Q-values (Equation 3.6) and minimized via backpropagation (Equation 3.7).

Training stability in DQN has been further improved through several enhancements. Between them, Experience replay introduces a buffer where past transitions are stored, allowing the agent to sample mini-batches of experiences uniformly or according to priority, thus making the process more stable and prone to converge [132]. Additionally, a separate target network could also be employed to stabilize updates, with its parameters updated less frequently or through Polyak [133]. Finally, several variants have been proposed to mitigate the reward overestimation bias of the algorithm [134],

including Double-DQN and Dueling-DQN that have become widely adopted [135], [136].

Despite being one of the earliest DRL methods, DQN remains a reference point in the literature. Its simplicity, effectiveness, and extensibility have made it a cornerstone for a wide range of RL applications, including those involving orchestration in dynamic and large-scale CC environments.

3.2.2 Trust Region Policy Optimization and Proximal Policy Optimization

Unlike the off-policy nature of DQN, Trust Region Policy Optimization (TRPO) is an on-policy RL algorithm that addresses the stability issues of policy-gradient methods. Its main objective is to determine an optimal step size for policy updates, thereby improving both convergence speed and robustness [137]. TRPO introduces the concept of a *trust region*, which defines the highest accepted distance between the updated and the previous policy. By enforcing this bound, the algorithm mitigates the risks of performance collapse and sample inefficiency often observed in traditional policy-gradient approaches. To do so, it formally defines the following surrogate objective:

$$\max_{\theta} \mathbb{E}_t \left[\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|A_t)} \hat{A}^{\pi_{\theta_{old}}}(S_t, A_t) \right] \tag{3.8}$$

where θ_{old} represents the vector of policy parameters before the update and $\hat{A}^{\pi_{\theta_{old}}}$ is an estimator of the advantage function from the older policy $\pi_{\theta_{old}}$. Along with this surrogate objective function, applying specific constraints to this equation, i.e., bounding the Kullback–Leibler divergence between π_θ and $\pi_{\theta_{old}}$, ensures monotonic policy improvement and allows, under certain conditions, the reuse of off-policy data [138].

Although theoretically efficient, TRPO requires solving a constrained optimization problem at every update step, which can become computationally demanding. While penalty-based reformulations exist, they introduce the practical challenge of selecting appropriate penalty coefficients, which is often non-trivial. To address these drawbacks, PPO was introduced as a simpler and more computationally efficient alternative [139]. So far, two main variants have been formalized and mostly used in the literature so far: PPO-Penalty and PPO-Clip [140], [141]. The first optimizes a regularized version of Equation (3.8), introducing an adaptive regularization parameter λ that depends on π_θ. On the other hand, the latter calculates a clipped version of the term:

$$\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)}$$

and considers as a learning objective the minimum between the clipped and the non-clipped versions. This prevents excessively large updates that could destabilize training while maintaining sufficient flexibility for exploration and improvement.

More specifically, PPO-Clip leverages a modified version of the surrogate function in Equation (3.8) as follows:

$$\max_{\theta} \mathbb{E}_t[\min(\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|A_t)}\hat{A}^{\pi_{\theta_{old}}}(S_t, A_t), clip(\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|A_t)}, 1-\epsilon, 1+\epsilon)\hat{A}^{\pi_{\theta_{old}}}(S_t, A_t))] \tag{3.9}$$

where function $clip(x, 1-\epsilon, 1+\epsilon)$ clips x within the interval $[1-\epsilon, 1+\epsilon]$, with ϵ being a hyperparameter that defines the clipping neighborhood.

Thanks to a more elegant and computationally efficient behavior than TRPO, PPO is a particularly interesting solution for DRL applications. PPO-Clip is arguably the most-interesting variant of PPO: it has proven to be remarkably simple and stable and to work consistently well in a wide range of scenarios, outperforming other algorithms, such as Advantage Actor–Critic. Its practical reliability and reduced computational overhead make it one of the most widely adopted policy-gradient algorithms for DRL, and a natural choice for investigating orchestration strategies in the CC.

3.2.3 Deep Sets Integration

DS represent a neural network architecture specifically designed to process sets as input [142], thus addressing the challenges associated with their unordered and variable sizes. Unlike traditional neural networks, which typically expect inputs to be ordered and of fixed size, DS architectures can naturally handle inputs of arbitrary cardinality without being affected by their ordering. Formally, given an input set of c elements, $X = \{x_1, \ldots, x_c\}$, the goal is to design neural networks that are either **Permutation-Invariant (PI)** or **Permutation-Equivariant (PE)** with respect to the ordering of the elements of the set. A function f is PI if $f(X) = f(p(X))$ for any permutation p, which means that the output is independent of the ordering of the input set. In contrast, a function f is considered PE if $f(p(X)) = p(f(X))$, implying that permutations of the input are reflected in the output.

In the context of orchestration within the CC, DS architectures provide several advantages:

- **Permutation handling**: Functions can be designed as PI or PE, ensuring that the learned model is not constrained by arbitrary cluster indexing.
- **Scalability**: The computational complexity of DS scales linearly with the number of clusters, with operations that can be easily parallelized over the elements of the set.
- **Flexibility**: DS can process sets of varying size, enabling generalization to environments with a changing number of clusters.

Mathematically, a DS network for PE can be expressed as:

$$\sigma(x\Lambda - 1\Gamma maxpool(x)) \tag{3.10}$$

where $x \in \mathbb{R}^{c \times m}$ represents the input features, $\Lambda, \Gamma \in \mathbb{R}^{m \times k}$ are trainable parameters, and $\sigma()$ is a non-linear activation function (e.g., ReLU). This formulation can be also considered PE with respect to the rows of x. Transformation $x\Lambda$ applies the same linear mapping to each row of x, while the max-pooling operation extracts the maximum value from each column, ensuring permutation invariance [143]. Stacking multiple layers of this form builds PE neural networks. Importantly, the operations are independent of c: once trained, DS networks can perform inference on sets of arbitrary size, with computational complexity scaling linearly with the number of elements in the set and parallelizable over the set elements, similar to batching in deep neural networks.

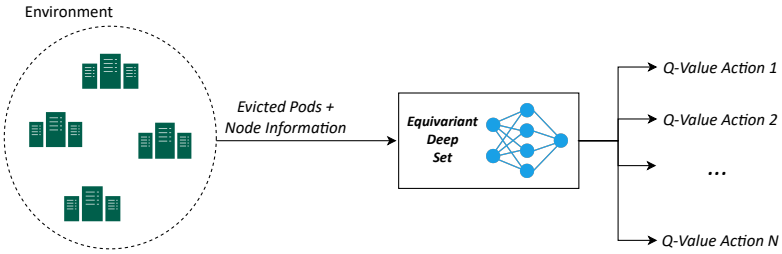


Figure 3.3: DQN-DS Integration

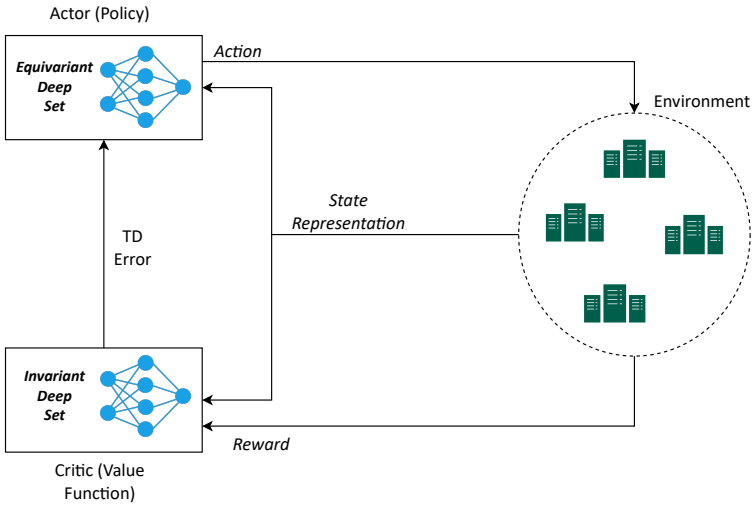


Figure 3.4: PPO-DS Integration

Figures 3.3 and 3.4 show how DS are integrated into the traditional DQN and PPO architectures. In the PPO-DS configuration, a stack of equivariant DS layers is employed to implement the Actor, while an invariant DS serves as the Critic. In the case of DQN-DS, an equivariant DS is used to approximate the Q-function.

3.3 Multi-Objective Optimization

To satisfy latency and computing requirements, orchestration solutions must be able to handle a wide range of microservices among a plethora of computing resources distributed among the CC. The diverse requirements of these microservices not only increase the computational demands of the orchestration problem but also make it challenging to balance conflicting objectives, such as minimizing deployment costs while meeting latency constraints, as the ideal trade-offs vary across use cases. Considering that each application may be sensitive to specific performance aspects, such as latency or deployment cost, it is challenging to specify precise preferences among different objectives in advance using widely proposed solutions, such as weighted sum or lexicographic approaches. However, in practice, such preferences may not be known in advance or may evolve over time, limiting the practical effectiveness of these traditional approaches. Therefore, a MO formulation is often necessary since the complex and unpredictable interactions between heterogeneous objective functions make it challenging, if not impossible, to define precise and explicit preferences in advance. In practice, when dealing with multiple competing objectives, it is desirable to present to the decision-maker a set of multiple "optimal" trade-offs, such that well-informed decisions can be made a posteriori [63], [64].

Formally, Multi-Objective Optimization (MOO) addresses optimization problems with two or more conflicting objective functions that need to be minimized or maximized simultaneously, seeking to find multiple solutions that balance the trade-offs between these objectives. Specifically, a MO-ILP problem can be generally written as:

$$\min \quad \mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_k(\mathbf{x})) \tag{3.11}$$

$$\text{s.t.} \quad \mathbf{A}\mathbf{x} \leq \mathbf{b} \tag{3.12}$$

$$\mathbf{x} \in \mathbb{Z}^n \tag{3.13}$$

where $(f_1, f_2, \ldots, f_k)$ are the objective functions, $\mathbf{x}$ are known as the decision variables, and $\mathbf{A}$, $\mathbf{b}$ are the constraint matrix and the right-hand-side coefficients, respectively.

In contrast to single-objective optimization, a MO-ILP has, in general, several "optimal" solutions, each achieving a different trade-off between the objectives. The set of optimal solutions to the MO-ILP consists of its PF, which is defined as:

$$\mathcal{PF} = \{\mathbf{x} \in \Omega \mid \mathbf{f}(\mathbf{x}) \succ_P \mathbf{f}(\mathbf{x}') \quad \forall \mathbf{x}' \in \Omega\}, \tag{3.14}$$

where $\mathbf{\Omega}$ is the feasible set of the MO-ILP, and $\succ_P$ is the Pareto dominance relation, defined as:

$$\mathbf{x} \succ_P \mathbf{x}' \iff (f_i(\mathbf{x}) \geq f_i(\mathbf{x}')\ \forall i) \wedge (\exists i\colon f_i(\mathbf{x}) > f_i(\mathbf{x}')) \tag{3.15}$$

In other words, a feasible solution $\mathbf{x}$ Pareto-dominates $\mathbf{x}'$ if and only if $\mathbf{x}$ improves over $\mathbf{x}'$ in at least one objective while not worsening the others [144].

Deriving the full PF is significantly more complex than conventional single-objective optimization, as it requires purposely designed solving algorithms. To address the complexity of MOO, a simple approach is to scalarize the objectives into a single one via a scalarization function $s\colon \mathbb{R}^k \to \mathbb{R}$. A popular approach for MO-ILP problems is defining s as a convex combination of objectives, such that the linearity of the objective function is preserved, and thus the problem can be solved as a single-objective ILP with conventional algorithms. According to this approach, each objective is assigned a scalar weight, whose value represents the importance of the specific objective function among the other. However, there are two main drawbacks to this approach: *1)* it requires specifying the relative importance of the objectives a priori since the interactions between the objectives are often nonlinear, meaning that predicting the effect of changing a coefficient in the resulting solution is impossible; *2)* linear combinations can only achieve solutions lying on the convex hull of the PF, which can result in missing many potentially interesting trade-offs.

For the above reason, proper MOO algorithms aim to either enumerate or approximate the PF, such that the decision-maker is presented with multiple solutions, and is then able to decide a posteriori on the trade-off that best satisfies the system's operational requirements. A popular and effective approach is leveraging conventional single-objective ILP solvers to either enumerate or approximate the PF [145]–[147]. Intuitively, the idea is to solve a sequence of suitable single-objective ILP problems, such that each solver call returns a Pareto-optimal solution. A classical and simple example of these strategies is the ε-constraint method [145], which optimizes one objective at a time while constraining all other objectives to be less or equal to a constant value ε. The algorithm can return a good approximation of the PF by choosing suitable discretization strategies for the objectives. Modern exact and approximating algorithm, e.g., [146], [147], refine this general methodology to further reduce the number of solver calls. Unfortunately, even for small-scale single-objective instances, state-of-the-art solvers such as HiGHS [148] can take hundreds of seconds to derive optimal solutions.

3.3.1 Multi-Objective Evolutionary Algorithms

To address these shortcomings, Evolutionary Algorithms (EAs) are a popular and effective solution for approximately solving MO problems. These algorithms combine conventional crossover and mutation operators of classical EAs with ad-hoc heuristics

for maximizing the diversity of the discovered PF [149]. Some of the most popular and effective algorithms employed in this thesis are the following:

- **Non-Dominated Sorting Genetic Algorithm (NSGA)-II** [150]: Born as an improved version of NSGA [151], it is characterized by three core design elements: i) a fast sorting procedure to identify Pareto-dominating solutions in short computational times, ii) the introduction of a crowding distance metric to maximize the spread of the solutions over the PF, and iii) an elitist approach, which preserves the best-found Pareto-dominating solutions at each iteration. The general applicability and effectiveness of NSGA-II make it a de-facto default choice for MO problems.

- **NSGA-III** [152]: improves over [150] for MO problems with a large number of objectives. To maximize the approximated diversity of the PF, instead of computing a crowding distance as in NSGA-II, it assigns each solution to a reference point in a normalized objective space, which the algorithm tries to cover uniformly. For high dimensions (i.e., 3 or more objectives), this procedure is more effective and computationally lighter than the crowding distance.

- **PSO** [153]: In addition to the characteristics of its architecture as an evolutionary algorithm, PSO is particularly suitable for multi-objective optimization due to its notable convergence speed and efficient exploration of the solution space. The standard approach for MO with PSO algorithms relies on an external repository to store non-dominated feasible solutions and use them to guide the evolution process. Over the years, many adaptations have been made to its structure, such as the selection methodologies to find local and global bests, the mutation operators [154], and the number of total swarms involved [155], [156]. In particular, the last aspect demonstrated to be particularly promising thanks to its ability to explore the solution space efficiently and the possibility of delegating every individual objective of the problem to a specific swarm. For this reason, a custom **MPSO** has been developed, inspired by the work described in [156]. Specifically, this implementation creates a swarm with a fixed number of particles for each objective and adopts strategies to influence the movement of the particles (i.e., the exploration of the solution space) tailored for dealing with integer solutions, such as Integer Polynomial mutation and a custom comparator to select dominated solutions while taking into account penalties related to constraint violations. This custom operators are also adopted for NSGA-II and NSGA-III implementations.

3.4 Chapter Summary

This chapter provided a detailed breakdown of the theoretical aspects behind both CI and RL methodologies, discussing their pros and cons when employed in resource and service management in CC environments. Section 3.1 delved into the main principles of the CI, which enable an effective implementation of algorithms such as GA, PSO with its variants, and GWO. Each one of them was discussed in-depth, including pseudo-algorithms, main parameters, and potential in dynamic scenarios such as the CC. Section 3.2 examined the concepts that serve as the foundation of RL methodologies. A quick overview of two of the most known algorithms in the current literature, DQN and PPO, was made. Furthermore, an integration with DS neural networks was defined. In the following sections, experimental results obtained demonstrate the significant benefits introduced by this paradigm, in particular regarding permutation handling, scalability, and flexibility. Finally, Section 3.3 explored the MOO, with a focus on the main EAs and their potential to provide multiple trade-offs to decision makers, allowing for an informed and a posteriori decision according to specific needs.

Chapter 4

Reinforcement Learning and Computational Intelligence for Compute Continuum Orchestration

As discussed previously, metaheuristic-based methods offer strong capabilities in exploring large parameter spaces and generating near-optimal configurations, while being less affected by inefficient sampling. RL, on the other hand, emphasizes autonomous learning and adaptation, though typically at the cost of higher sample inefficiency. As a result, RL is particularly advantageous in environments characterized by strong dynamicity and sudden variations, where metaheuristic methods can struggle and often require retraining to maintain performance.

To evaluate the relative strengths of these two paradigms in the context of service orchestration and management in the CC, this chapter presents several use cases which have been defined and evaluated using different approaches capable of reproducing services operating within a CC environment.

4.1 Comparing Service Management Approaches for the Compute Continuum

Modern distributed computing environments, thanks to the advent of several enabling technologies such as MEC, effectively represent a CC environment, a capillary network of computing resources that extend from the edge of the network to the cloud, which enables a dynamic and adaptive service fabric. Efficient coordination of resource allocation, exploitation, and management in the CC represents quite a challenge, which has stimulated researchers to investigate innovative solutions based on smart techniques

such as RL and CI. This section makes a comparison of different optimization algorithms and a first investigation of how they can perform in this kind of scenario. Specifically, this comparison includes the DQN, PPO, GA, PSO, QPSO, MPSO, and the GWO[1].

To address service management in the CC, a first description of an optimization problem that aims to find the proper deployment for a pool of services of different importance has been defined. Specifically, the aim is to activate multiple instances of these services on the resources available throughout the continuum. Once these instances are activated on a proper device, they will start processing requests. To enrich the evaluation of the performance of a given deployment configuration, a metric that measures the ratio between the number of user requests that were successfully executed and the total number of requests that were generated in a given time window t, called PSR, has been defined.

Specifically, $\mathbb{C} = \{c_1, c_2, \ldots, c_n\}$ is the set of application components that must be allocated by the infrastructure provider in the CC. Each application component has fixed resource requirements a_{res} measured as the number of CPU or GPU cores that should be available for processing on servers. For simplicity, the resource requirements for each application $c_i \in C$ are immutable. This assumption is consistent with the related literature [157] and also with modern container-based orchestration techniques, which allow users to specify the number of CPU and GPU cores to assign to each container. Each application instance is modeled as an independent M/M/1 First-Come First-Served (FCFS) queue that processes requests in a sequential fashion. In addition, queues have a maximum buffer size, i.e., queues can buffer up to a maximum number of requests. As soon as the buffer is full, the queue will start dropping incoming requests. Computing resources are defined with the set of devices $\mathbb{D} = \{d_1, d_2, \ldots, d_n\}$, where each $d_i^k \in D$ has an associated type $k \in \{\text{Cloud, Fog, Edge}\}$ to describe the characteristics and location of the server. In addition, each device $d_i \in \mathbb{D}$ is assigned with D_i^{res} resources, where res represents the number of computing CPU or GPU cores. Moreover, servers at the same computing layer would have equal computing capacity; thus, servers at the upper computing layers would have higher capacity. At a given time t, the number of resources requested by applications allocated to servers cannot exceed the servers' capacity D_i^{res}.

Taking into account the perspective of the service provider, it needs to find a deployment for the application components $c \in C$ that maximizes performance, which can be formally defined as follows:

$$argmax_{C,D} f(x) \tag{4.1}$$

[1]F. Poltronieri, C. Stefanelli, M. Tortonesi, and M. Zaccarini, "Reinforcement learning vs. computational intelligence: Comparing service management approaches for the cloud continuum," Future Internet, vol. 15, no. 11, 2023. doi:10.3390/fi15110359.

where C represents the applications that need to be deployed, D is the set of devices where to allocate applications, and x is the deployment of the service component. Additionally, the objective function $f(x)$ is defined as follows:

$$f(x) = \sum_{k=1}^{n} \Theta_k \times PSR(c_k, x) \tag{4.2}$$

where each Θ_i component is a weight factor that identifies the criticality of a specific application $c_i \in C$ and $PSR(c_i, x)$ is the PSR for application c_i using configuration x.

It is the responsibility of the infrastructure provider to select proper values for the Θ_i components, which represent the utility that an infrastructure provider gains by running a particular application component of type c_i and remain fixed at a certain time t. Although it is a relatively simple approach, it could be reasonably effective to treat several service components with different priorities. To maximize the value of (4.2), the infrastructure provider needs to improve the current service deployment configuration so that the percentage of satisfied requests of the most-important services is prioritized.

To solve the above service management problem, how the instances of applications $c_i \in C$ are deployed on the device $d \in D$ is represented by an array-like service configuration with integer values, inspired by the one presented in [158] as follows:

$$SC = \{X_{c_1,d_1}, X_{c_2,d_1}, \ldots, X_{c_{k-1},d_{n-1}}, \ldots, X_{c_k,d_n}\} \tag{4.3}$$

where the value of the element X_{c_i,d_j} describes the number of application components of type c_i that are allocated on device d_j to process requests' application c_i, n is the number of devices, and k the $|C|$.

Concerning the application of PPO and DQN to the service management problem, the MDP problems defined are slightly different, one specific to the DQN and the other to PPO. Both define a set of states S, each defined as the above deployment array (4.3) to represent the allocation of service components on devices $X_{c_i,d_j} \in C$, and a reward function R, which is the immediate reward $R_a(S, S')$ received after performing an action $a \in A$ in a state s. The main difference is related to how the deployment array in (4.3) is analyzed and, therefore, the actions that the agent can perform. For the DQN MDP, the agent has to analyze each element of the deployment array sequentially. Specifically, at each timestamp t_i, the DQN agent analyzes an element of the deployment array $SC[t_i]$, starting from the beginning to the end. When analyzing $SC[t_i]$, the agent performs an action $a \in A$ on the element $SC[t_i]$ to modify the active instances for the application components c_i on d_j, the value of $X_{c_i,d_j} \in C$. To do so, the agent can either (i) do nothing, (ii) activate up to two new instances, or (iii) deactivate up to two instances, where each of these actions is encoded through an integer value in $[0, 5]$. Instead, for PPO, A has the shape of a multi-discrete action space, i.e., a vector that extends the discrete action space over a space of independent discrete actions [159], which consists of two vectors of choices: to perform the first choice, the agent picks an

element of C, corresponding to the number of active instances for service component c_i on device d_j. Then, for the second choice, the agent modifies the number of active instances to improve the current allocation according to the three actions described for the DQN. Therefore, according to this formulation, the PPO agent does not scan the deployment array in a sequential fashion; instead, it can learn a smarter way to improve the overall value of Equation (4.2).

Concerning the reward definition, it is modeled as the difference in Equation (4.2) calculated between two consecutive time steps. Specifically, this reward function verifies whether a specific action can improve or not the value of Equation (4.2).

$$R_a(S, S') = R *_a (S, S') - \Phi \tag{4.4}$$

Specifically, $R *_a (S, S')$ and Φ are defined as:

$$R *_a (S, S') = f(S') - f(S) \tag{4.5}$$

$$\Phi = \#infeasible * \gamma \tag{4.6}$$

where $f(S')$ and $f(S)$ are the objective functions calculated, respectively, in states S' and S and Φ is the penalty quantified in the number of accumulated infeasible allocations during the simulation multiplied by a facto γ, which we set to 0.1. If there is an improvement from one time step to the next one ($R *_a (S, S') > 0$), the agent receives an additional bonus of 3.0 as compensation for its profitable move. Otherwise, the action taken is registered as a wrong pass, since it is not remunerative in improving the objective function calculated previously. If the agent reaches an amount of 150 wrong passes, the training episode terminates immediately and resets the environment to the initial state.

On the other end, CI techniques require their adopters to define a "fitness function" or "evaluation function" to drive the optimization process. One of the advantages of these techniques is that it is possible to use the objective function directly as the fitness function. However, it is common to add additional components (e.g., a penalty component) to guide the optimization process to better solutions. In order to consider the resource limitations imposed by the scenario, two different CI configurations are adopted: a baseline configuration, namely "GA", "PSO", "QPSO", "MPSO", and "GWO", which uses as the fitness function the objective function $f(x)$, and another configuration called Enforced ConstrainT (ECT), namely "GA-ECT", "PSO-ECT", "QPSO-ECT", "MPSO-ECT", and "GWO-ECT", which instead takes into account the infeasible allocations generated at a given iteration as a penalty in the fitness function $J(x)$:

$$J(x) = f(x) - \Phi \tag{4.7}$$

where $f(x)$ is the target function (i.e., the problem objective) and Φ is the penalty component visible in Equation (4.6). Unlike the baseline configuration, the ECT configuration would also minimize the number of infeasible allocations while maximizing the PSR. Finally, the ECT approach reconstructs the operating conditions of the RL algorithms and forces both metaheuristics to respond to the same challenge, thus allowing a fair comparison with the DQN and PPO.

To compare the CI and RL approaches for service management, a use case for a simulator capable of reenacting services running in a CC scenario has been defined. This simulator is built by extending the Phileas simulator [160], a discrete event simulator designed to reenact VoI-based services in Fog computing environments. However, it represented a good commodity to evaluate different optimization approaches for service management in the CC. In fact, Phileas allows to accurately simulate the processing of service requests on top of a plethora of computing devices with heterogeneous resources and to model the communication latency between the parties involved in the simulation, i.e., from users to computing devices and vice versa. Its VoI potential in CC environments are discussed in depth in the next section. This comparison evaluates the quality of the best solutions generated with respect to the objective function (4.2), but also in terms of sample efficiency, i.e., the number of evaluations of the objective function. Finally, a simulation of a sudden disconnection of computing resources evaluates whether these approaches can work properly when environmental conditions change.

The scenario simulated through Phileas describes a smart city that provides several applications to its citizens. Specifically, the use case contains the description of a total of 13 devices distributed among the CC: 10 Edge devices, 2 Fog devices, and a Cloud device with unlimited resources to simulate unlimited scalability. Along with the devices' description, the use case defines 6 different smart city applications, namely: healthcare, pollution-monitoring, traffic-monitoring, video-processing, safety, and audio-processing applications, whose importance is described by the $Theta$ parameters shown in Table 4.1.

Table 4.1: Service description: time between request generation and compute latency modeled as exponential random variables with rate parameter λ for each service type and resource occupancy.

Service Type	**Weight (θ)**	**Time Between Req. Gen. (λ)**	**Compute Latency (λ)**	**Resource Consumption**
Healthcare	1.0	1/120 (ms)	1/150 (ms)	4.5 (cores)
Pollution	0.5	1/45 (ms)	1/250 (ms)	3.5 (cores)
Traffic	0.5	1/40 (ms)	1/300 (ms)	3.5 (cores)
Video	1.0	1/100 (ms)	1/225 (ms)	6.0 (cores)
Safety	1.0	1/100 (ms)	1/150 (ms)	4.0 (cores)
Audio	0.5	1/25 (ms)	1/200 (ms)	3.0 (cores)

To simulate a workload for the described applications, 10 different groups of users located at the Edge generate requests according to the configuration values illustrated in Table 4.1. The request generation is a stochastic process modeled using 10 different random variables with an exponential distribution, i.e., one for each user group. Furthermore, Table 4.1 also reports the computed latency for each service type. As for the time between message generation, the processing time of a task has been modeled by sampling from a random variable with an exponential distribution. The "compute latency" value does not include queuing time, i.e., the time that a request takes before being processed. Finally, the resource occupancy indicates the number of cores that each service instance requires to be allocated on a computing device. The simulation is set to be 10 s long, including 1 s of warmup, to simulate the processing of approximately 133 requests per second. Finally, Table 4.2 reports the intra-layer communication model, where each element is the configuration for a normal random variable used to simulate the transfer time between the different layers of the CC.

Table 4.2: Communication latency configuration for the Cloud Continuum use case.

Layers	**Edge**	**Fog**	**Cloud**
Edge	$\mu = 5.00$ (ms) $\sigma = 3.00$ (ms)	$\mu = 15.00$ (ms) $\sigma = 5.00$ (ms)	$\mu = 100$ (ms) $\sigma = 6.00$ (ms)
Fog		$\mu = 5.00$(ms) $\sigma = 3.00$ (ms)	$\mu = 80.00$ (ms) $\sigma = 8.00$ (ms)
Cloud			$\mu = 17.00$ (ms) $\sigma = 7.00$ (ms)

Concerning optimization approaches, RL algorithms comprehend the open-source and state-of-the-art DQN and PPO algorithms provided by Stable Baselines3 [161]. On the other hand, GA, PSO, QPSO, MPSO, and GWO implementations are provided by a Ruby metaheuristic library called ruby-mhl [162].

To collect statically significant results from the evaluation of the CI approaches, the log of 30 optimization runs have been collected, where each optimization run consisted of 50 iterations of the metaheuristic algorithm. This ensured the interpretability of the results and that the use of different seeds can significantly change the outcome of the optimization process. At the end of the 30 optimization runs, the average best value found by each approach verified which one performed better in terms of the value of the objective function and sample efficiency.

In terms of the configuration details of each metaheuristic, GA setup considered a population of 128 randomly initialized individuals, an integer genotype space, a mutation operator implemented as an independent perturbation of all individual chromosomes sampled from a random variable of a geometric distribution with a probability of success of 50%, and an extended intermediate recombination operator controlled by

a random variable with a uniform distribution in [0.5, 1.5] [163]. Moreover, the lower and an upper bounds are set to 0 and 15, respectively. At each iteration, the GA generates the new population using a binary tournament selection mechanism, which applies the configured mutation and recombination. PSO is characterized by a swarm size of 40 randomly initialized individuals in the float search space [0, 15] and the acceleration coefficients C1 and C2 to 2.05. QPSO, had a swarm of 40 randomly initialized individuals in [0, 15] and a contraction–expansion coefficient α of 0.75. These particular parameter configurations have shown very promising results in different analyses made in the past [57], [164]. Considering GWO, the population consisted of 30 individuals and the same lower and upper bounds of 0 and 15. Finally, MPSO considered an initial number of swarms to 4, each with 50 individuals, and the maximum number of non-converging swarms to 3.

Regarding the DRL algorithms, the DQN scans the entire allocation array sequentially twice for 156 time steps, while PPO uses a maximum of 200 time steps, which correspond to their respective episode length. Since the DQN implementation of Stable Baselines3 does not support the multi-discrete action space as PPO does, we chose this training model to ensure the training conditions were as similar as possible for both algorithms. During a training episode, the agent modifies how service instances are allocated on top of the CC resources. Concerning the DRL configurations, the neural network architecture setup includes 2 fully connected layers with 64 units each and a Rectified Linear Unit activation function for the DQN and PPO. Furthermore, both the DQN and PPO pass through a training period of 100, 000 time steps long. Then, to collect statistically significant results comparable to the ones of the CI algorithms, both trained models were tested 30 times.

Figure 4.1 reports the average reward during the training process obtained by PPO and DQN. Both algorithms showed a progression in terms of average reward during the training process. Furthermore, Figure 4.1 shows that the average reward converged to a stable value before the end of the training process, thus confirming the validity of the reward structure presented. In this regard, PPO showed a better reward improvement, reaching a maximum near 200. Differently, the DQN remained stuck in negative values despite a rapid reward increase (around Iteration 50000) during the training session.

Aside from the DRL training, Figure 4.2 shows the convergence process of the CI algorithms, which is an illustrative snapshot of the progress of the optimization process. Specifically, it shows the convergence process of one of the 30 optimization runs and visualizes the GA, PSO, QPSO, MPSO, and GWO along with their related ECT versions. For all traditional metaheuristics displayed, it is easy to notice how they can converge very quickly. Instead, considering their ECT variants, the GA, PSO, and GWO struggled a bit to deal with their imposed constraints. In contrast, QPSO-ECT and MPSO-ECT demonstrated very similar performance compared to the previous case. In particular, these two algorithms did not appear to be negatively affected by the introduction of the penalty factor, and they were still able to find the best solution overall

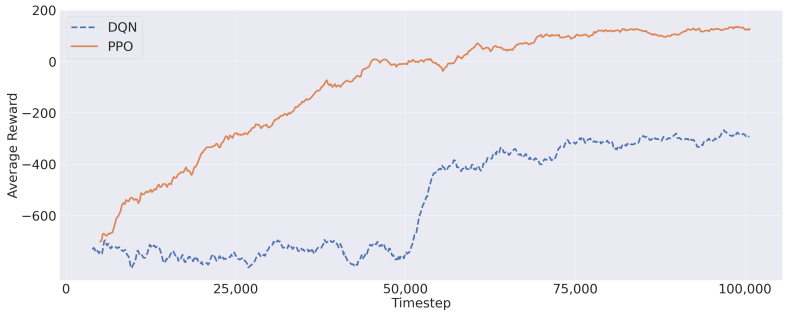
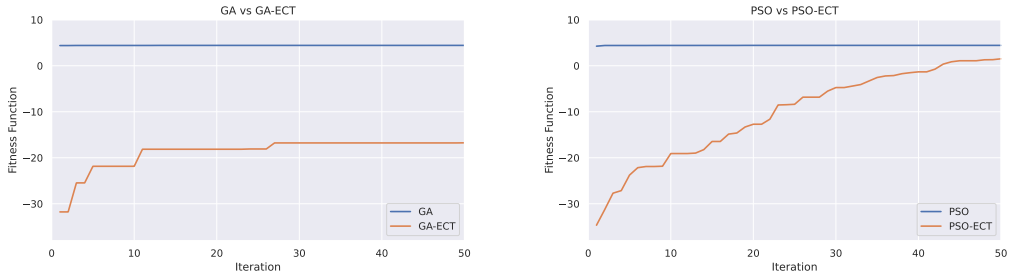


Figure 4.1: The DQN and PPO mean reward during the training process.

without significant difficulty. In this regard, MPSO makes use of four large swarms of 50 particles, i.e., at each iteration, the number of evaluations of the objective function was even larger than GA, which was configured with 128 individuals.

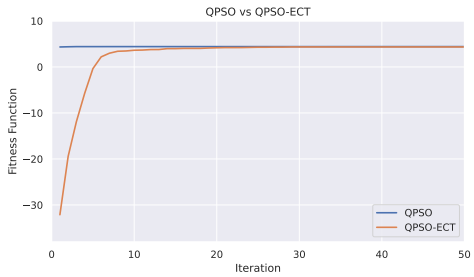
To give a complete summary of the performance of the adopted methodologies, Table 4.3 the average and standard deviation collected during the 30 optimization runs. More specifically, it reports the results related to the objective function, the number of generations needed to find the best objective function, and the average number of infeasible allocations associated with the best solutions found. The "Sample efficiency" column represents the average number of samples that were evaluated by CI algorithms in order to find the best solution. Specifically, this was approximated by multiplying the number of average generations with the number of samples evaluated at each generation, e.g, the number of samples at each generation for GA would be 128. Instead, for the DRL algorithms, the "Sample Efficiency" represents the average number of steps, i.e., an evaluation of the objective function that the agent took to achieve the best result, in terms of the objective function, during the 30 episodes. Regarding the objective function, Table 4.3 shows that MPSO was the algorithm that achieved the highest average score for this specific experiment, as opposed to the GA-ECT, which confirmed the poor trend shown in Figure 4.2. However, the two versions of MPSO required the largest number of samples to achieve high-quality solutions. It is important to note that all ECT algorithms delivered great solutions in terms of balancing the objective function against the number of infeasible allocations. This trend highlights that the use of a penalty component to guide the optimization process of CI algorithms can provide higher efficiency in exploring the search space.

On the other hand, the RL algorithms exhibited competitive results compared to the CI algorithms in terms of maximizing the objective function. Specifically, PPO was the fastest to reach its best result, with a great objective function along with a particularly low number of infeasible allocations (the lowest if we exclude the ECT variants). The DQN was demonstrated to be not as effective as the PPO. Despite requiring the

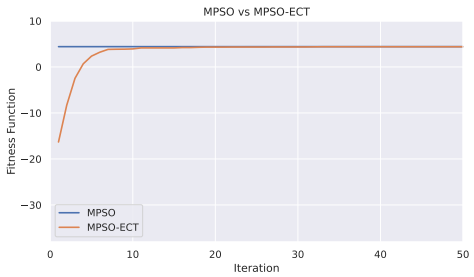


(a) GA vs GA-ECT

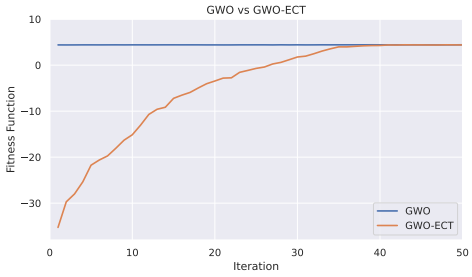
(b) PSO vs PSO-ECT



(c) QPSO vs QPSO-ECT



(d) MPSO vs MPSO-ECT



(e) GWO vs GWO-ECT

Figure 4.2: Comparison of each metaheuristic (baseline) against its constrained ECT variant across iterations. The constrained versions incorporate a penalty component in the fitness function.

second fewest generations to find its best solution, all metaheuristic implementations in this analysis outperformed it in both the objective function value and total infeasible allocations. The main reason behind this could lie in the implementation of the DQN provided by Stable Baselines3, which does not integrate any prioritized experience replay or improved versions like the Double-DQN. Consequently, it was not capable of dealing with more complex observation spaces, such as the multi-discrete action space of PPO. With a complex problem such as the one we are dealing with in this manuscript, the Stable Baseline3 DQN implementation appeared to lack the tools to

Table 4.3: Comparison of the average best solutions and the sample efficiency for the chosen algorithms.

Algorithm	Objective Function		Generation		Sample Efficiency	Infeasible Allocation	
	Avg	Std	Avg	Std		Avg	Std
PPO	4.1004	0.1567	–	–	**21.43 (18)**	18.83	6.39
DQN	3.7895	0.51762	–	–	24.17 (40.95)	448.43	83.27
GA	4.4288	**0.0027**	**28.07**	14.9088	3593	476.27	34.20
PSO	4.3221	0.0051	34.73	10.4977	1389	425.67	38.67
QPSO	4.4255	0.0074	30.9	14.1941	1236	482.53	204.10
MPSO	**4.4313**	0.0060	32.3667	12.7617	6473	406.07	209.89
GWO	4.4279	0.00600	31.17	14.1326	935	287.03	32.66
GA-ECT	4.1467	0.2829	39.7	8.0049	5082	204.40	11.83
PSO-ECT	4.3325	0.1014	49.57	**0.6789**	1983	31.97	8.28
QPSO-ECT	4.3589	0.0560	44.63	6.4513	1785	**0.0**	**0.0**
MPSO-ECT	4.3981	0.0254	41.9	7.8052	8380	**0.0**	**0.0**
GWO-ECT	4.3856	0.0324	46.9	2.3540	1407	**0.0**	**0.0**

achieve the same performance as PPO.

The last step of this experimentation was to analyze the performance of the best solutions presented previously, showing how the different solutions perform in terms of the PSR and average latency. Even if the problem formulation does not take into account latency minimization, it is still interesting to analyze how the various algorithms can distribute the service load across the CC and to see which offers the highest quality solution. It is expected that solutions that make use of edge and fog computing devices should be able to reduce overall latency. However, given the limited computing resources, there is a need to take advantage of the cloud layer to deploy service instances. Specifically, for each algorithm and each measure, Tables 4.4, 4.5, and 4.6 report both the average and the standard deviation of the best solutions found during the 30 optimization runs grouped by service. From these data, it is easy to observe that most metaheuristic approaches can find an allocation that nearly maximizes the PSR of the mission-critical services (identified in healthcare, video, and safety) and the other as well. Contrarily, both DRL approaches cannot reach PSR performance as competitively as the CI methodologies. This aspect explains why their objective functions visible in Table 4.3 ranked among the lowest. Nevertheless, both PPO and the DQN provided very good results in terms of the average latency for each microservice. Despite certain shortcomings in the PSR of specific services, particularly Audio and Video, PPO consistently outperformed most of them in terms of latency.

On the other hand, looking at the average service latency of the other approaches,

Table 4.4: Comparison of the average PSR and latency of the RL best solutions; the standard deviation is enclosed in ().

Algorithm	Service	PSR	Latency (ms)	Algorithm	Service	PSR	Latency (ms)
PPO	pollution	**0.93 (0.18)**	41.69 (37.57)	DQN	pollution	**1.0 (0.0)**	**41.56 (21.94)**
	traffic	0.63 (0.26)	**22.39 (20.02)**		traffic	0.92 (0.07)	62.04 (38.36)
	video	0.47 (0.25)	34.65 (30.15)		video	0.71 (0.33)	104.14 (63.44)
	audio	0.48 (0.23)	34.92 (29.22)		audio	0.57 (0.33)	94.37 (69.74)
	healthcare	0.82 (0.19)	46.03 (39.38)		healthcare	0.71 (0.39)	131.04 (83.04)
	safety	0.68 (0.34)	27.036 (23.11)		safety	0.66 (0.39)	128.98 (84.51)

Table 4.6 shows that the algorithms performed quite differently. Indeed, it is clear how the ECT methodologies consistently outperformed their counterparts in the majority of cases. Among them, QPSO-ECT emerged as the most-efficient overall in minimizing the average latency, particularly for mission-critical services. Specifically, QPSO-ECT overcame the GA-ECT by an average of 50%, PSO-ECT by 37%, GWO-ECT by 48%, and MPSO-ECT by 19% for these services. Oppositely, both variants of the GA registered the worst performance, with a significant number of micro-services registering a latency between 100 and 200 ms. However, minimizing the average service latency was not within the scope of this experiment, which instead aimed at maximizing the PSR, as is visible in (4.2). To conclude, PPO emerged as the best DRL algorithm. It can find a competitive value in terms of the objective function along with the best results in terms of sample efficiency and latency at the price of a longer training procedure, when compared to the CI algorithms. While the ECT variants of the metaheuristics included in this comparison demonstrated great performance as well, they require much more samples to find the best solution.

The last part of this analysis consists in a what-if scenario analysis in which the cloud computing layer is suddenly deactivated to verify the effectiveness of DRL algorithms in dynamic environments. Therefore, the service instances that were previously running in the cloud need to be reallocated on the over devices available if there is enough resource availability. To generate a different allocation of service component that takes into account the modified availability of computing resources, the same models trained on the previous scenario have been tested for 30 episodes. Instead, for the CI algorithms,

Table 4.5: Comparison of the average PSR and latency of the best CI solutions; the standard deviation is enclosed in ().

Algorithm	Service	PSR	Latency (ms)	Algorithm	Service	PSR	Latency (ms)
GA	pollution	**1.0 (0.0)**	**36.54 (17.65)**	PSO	pollution	**1.0 (0.0)**	**48.60 (23.35)**
	traffic	0.97 (0.006)	91.65 (16.81)		traffic	0.98 (0.008)	84.54 (27.79)
	video	0.98 (0.003)	186.57 (14.37)		video	0.98 (0.0026)	175.39 (24.28)
	audio	0.96 (0.005)	181.02 (20.87)		audio	0.96 (0.008)	176.53 (21.54)
	healthcare	0.98 (0.003)	197.76 (7.6)		healthcare	0.98 (0.003)	194.75 (10.16)
	safety	0.99 (0.002)	200.08 (0.56)		safety	0.98 (0.002)	198.72 (7.04)
QPSO	pollution	**1.0 (0.0)**	**45.28 (23.72)**	MPSO	pollution	**1.0 (0.0)**	**44.52 (22.41)**
	traffic	0.97 (0.0084)	123.4 (60.42)		traffic	0.97 (0.009)	99.43 (55.46)
	video	0.98 (0.003)	180.8 (24.17)		video	0.98 (0.002)	170.50 (27.50)
	audio	0.96 (0.01)	173.38 (31.09)		audio	0.96 (0.007)	163.05 (40.85)
	healthcare	0.98 (0.003)	195.13 (13.83)		healthcare	0.98 (0.003)	193.47 (16.27)
	safety	0.98 (0.002)	199.69 (2.99)		safety	0.98 (0.002)	198.25 (8.62)
GWO	pollution	**1.0 (0.0)**	**40.32 (19.22)**				
	traffic	0.96 (0.012)	59.85 (27.93)				
	video	0.97 (0.006)	173.64 (14.83)				
	audio	0.96 (0.012)	177.2 (20.08)				
	healthcare	0.98 (0.003)	197.03 (6.13)				
	safety	0.98 (0.005)	197.47 (5.96)				

Table 4.6: Comparison of the average PSR and latency of the best CI-ECT solutions; the standard deviation is enclosed in ().

Algorithm	Service	PSR	Latency (ms)	Algorithm	Service	PSR	Latency (ms)
GA-ECT	pollution	**0.99 (0.008)**	**56.15 (12.72)**	PSO-ECT	pollution	**0.99 (0.01)**	89.26 (32.94)
	traffic	0.93 (0.05)	77.90 (32.26)		traffic	0.94 (0.027)	**82.74 (44.22)**
	video	0.91 (0.11)	139.46 (45.33)		video	0.94 (0.063)	130.18 (39.57)
	audio	0.84 (0.20)	143.04 (43.21)		audio	0.92 (0.095)	93.59 (47.11)
	healthcare	0.90 (0.21)	170.90 (45.81)		healthcare	0.97 (0.018)	123.02 (47.80)
	safety	0.93 (0.13)	183.71 (29.27)		safety	0.98 (0.009)	134.19 (49.64)
QPSO-ECT	pollution	**1.0 (0.0)**	53.55 (42.93)	GWO-ECT	pollution	**1.0 (0.0)**	98.51 (43.30)
	traffic	0.93 (0.078)	**52.32 (39.72)**		traffic	0.95 (0.02)	**85.09 (39.50)**
	video	0.95 (0.038)	92.17 (56.98)		video	0.97 (0.020)	143.97 (38.01)
	audio	0.93 (0.029)	68.20 (48.67)		audio	0.94 (0.028)	115.38 (38.88)
	healthcare	0.98 (0.010)	74.37 (44.87)		healthcare	0.98 (0.01)	109.57 (38.22)
	safety	0.98 (0.004)	78.01 (45.06)		safety	0.98 (0.006)	109.93 (39.57)
MPSO-ECT	pollution	**1.0 (0.0)**	62.63 (48.65)				
	traffic	0.96 (0.013)	**51.58 (33.52)**				
	video	0.97 (0.02)	127.30 (39.53)				
	audio	0.95 (0.025)	79.53 (41.70)				
	healthcare	0.98 (0.008)	89.20 (39.78)				
	safety	0.98 (0.006)	76.07 (45.58)				

a cold restart technique was used, consisting of running another 30 optimization runs, each with 50 iterations. This was done to ensure the statistical significance of these experiments. After the additional optimization runs, all CI algorithms should be able to find optimized allocations that consider the different availability of computing resources in the modified scenario, i.e., exploiting only the edge and fog layers.

Table 4.7: Comparison of the average best solutions and the sample efficiency for the chosen algorithms in the what-if scenario.

Algorithm	**Objective Function**		**Generation**		**Sample Efficiency**	**Infeasible Allocation**	
	Avg	**Std**	**Avg**	**Std**		**Avg**	**Std**
PPO	3.9320	0.1915	-	-	**36.27 (46.82)**	16.67	6.42
DQN	1.6071	0.3228	-	-	128.5 (25.37)	293.13	52.55
GA	3.4371	0.1398	**33.5**	11.0383	4288	398.87	11.04
PSO	4.1917	0.0927	46.0667	4.4793	1843	340.47	36.15
QPSO	4.3677	0.0159	37.0	10.017	1480	64.43	19.67
MPSO	**4.3771**	**0.0123**	37.16	8.3463	7432	66.40	18.53
GWO	4.0910	0.2003	48.2	3.4780	1446	230.33	32.38
GA-ECT	2.2287	0.4854	36.6633	10.2166	4693	199.97	30.69
PSO-ECT	3.9613	0.2053	49.33	**0.9222**	1973.20	33.90	14.16
QPSO-ECT	4.1593	0.1512	45.7333	4.1848	1829	**0.0**	**0.0**
MPSO-ECT	4.2681	0.0313	47.3333	7.4664	1493	**0.0**	**0.0**
GWO-ECT	4.1297	0.0727	48.9667	1.16078	1469	0.7	0.88

Table 4.8: Comparison of the average PSR and latency of the RL best solutions in the what-if scenario; the standard deviation is enclosed in ().

Algorithm	Service	PSR	Latency (ms)	Algorithm	Service	PSR	Latency (ms)
PPO	pollution	**0.91 (0.25)**	16.93 (8.9)	DQN	pollution	**1.0 (0.0)**	18.8 (3.55)
	traffic	0.60 (0.25)	14.09 (4.6)		traffic	0.93 (0.07)	18.75 (5.17)
	video	0.41 (0.24)	**13.39 (6.67)**		video	0.45 (0.25)	14.51 (7.99)
	audio	0.42 (0.22)	15.97 (6.95)		audio	0.22 (0.27)	6.76 (7.61)
	healthcare	0.76 (0.26)	19.99 (5.11)		healthcare	0.050 (0.13)	1.95 (5.07)
	safety	0.67 (0.35)	18.84 (8.4)		safety	0.0 (0.0)	**0.0 (0.0)**

Table 4.9: Comparison of the average PSR and latency of the best CI solutions in the what-if scenario; the standard deviation is enclosed in ().

Algorithm	Service	PSR	Latency (ms)	Algorithm	Service	PSR	Latency (ms)
GA	pollution	**1.0 (0.0)**	15.09 (2.9)	PSO	pollution	**1.0 (0.0)**	16.04 (5.07)
	traffic	0.89 (0.13)	15.57 (4.97)		traffic	0.95 (0.02)	**15.93 (4.6)**
	video	0.70 (0.15)	19.83 (5.94)		video	0.89 (0.06)	17.84 (5.27)
	audio	0.30 (0.28)	**13.18 (10.11)**		audio	0.86 (0.06)	22.13 (6.26)
	healthcare	0.84 (0.09)	28.60 (2.9)		healthcare	0.94 (0.03)	28.41 (3.74)
	safety	0.78 (0.18)	26.99 (4.83)		safety	0.95 (0.03)	28.31 (4.62)
QPSO	pollution	**0.99 (0.006)**	**17.02 (7.47)**	MPSO	pollution	**1.0 (0.0)**	16.85 (6.95)
	traffic	0.96 (0.014)	18.05 (4.93)		traffic	0.96 (0.013)	**16.57 (3.61)**
	video	0.96 (0.021)	18.71 (3.7)		video	0.96 (0.01)	18.44 (3.98)
	video	0.96 (0.021)	18.71 (3.73)		audio	0.93 (0.015)	19.24 (5.56)
	healthcare	0.98 (0.010)	26.26 (2.68)		healthcare	0.97 (0.007)	25.11 (3.005)
	safety	0.98 (0.007)	26.94 (3.76)		safety	0.98 (0.006)	27.48 (3.91)
GWO	pollution	**1.0 (0.0)**	15.12 (3.24)				
	traffic	0.94 (0.027)	**14.52 (4.24)**				
	video	0.86 (0.057)	17.53 (4.25)				
	audio	0.76 (0.204)	23.21 (5.38)				
	healthcare	0.92 (0.07)	28.29 (2.85)				
	safety	0.94 (0.031)	30.55 (2.02)				

Table 4.10: Comparison of the average PSR and latency of the best CI-ECT solutions in the what-if scenario; the standard deviation is enclosed in ().

Algorithm	Service	PSR	Latency (ms)	Algorithm	Service	PSR	Latency (ms)
GA-ECT	pollution	**1.0 (0.0)**	17.59 (4.27)	PSO-ECT	pollution	**0.99 (0.006)**	17.58 (5.46)
	traffic	0.92 (0.12)	19.51 (5.04)		traffic	0.88 (0.096)	17.43 (4.60)
	video	0.46 (0.24)	14.77 (7.65)		video	0.71 (0.197)	**16.12 (5.03)**
	audio	0.37 (0.30)	10.78 (9.73)		audio	0.83 (0.15)	19.30 (5.62)
	healthcare	0.35 (0.38)	12.22 (12.31)		healthcare	0.94 (0.055)	24.15 (3.12)
	safety	0.26 (0.35)	**9.79 (12.11)**		safety	0.95 (0.041)	25.41 (3.34)
QPSO-ECT	pollution	**0.99 (0.006)**	18.32 (7.58)	GWO-ECT	pollution	**0.98 (0.045)**	20.65 (5.85)
	traffic	0.93 (0.030)	17.82 (5.07)		traffic	0.87 (0.09)	16.76 (3.85)
	video	0.79 (0.17)	17.47 (4.99)		video	0.83 (0.06)	**16.51 (4.26)**
	audio	0.87 (0.06)	**18.06 (6.18)**		audio	0.86 (0.061)	18.12 (4.12)
	healthcare	0.97 (0.01)	23.52 (2.74)		healthcare	0.96 (0.026)	24.54 (2.08)
	safety	0.98 (0.007)	26.94 (3.76)		safety	0.98 (0.012)	24.83 (2.29)
MPSO-ECT	pollution	**1.0 (0.0)**	18.10 (7.76)				
	traffic	0.93 (0.03)	17.94 (4.34)				
	video	0.89 (0.03)	18.33 (4.39)				
	audio	0.89 (0.042)	**17.38 (4.95)**				
	healthcare	0.97 (0.010)	25.19 (3.42)				
	safety	0.98 (0.010)	24.54 (2.72)				

As for the previous experimental configuration, the statistics collected during op-

timization runs are reported to compare the best values of the objective function and the PSR of services in Tables 4.7, 4.8, 4.9, and 4.10. Looking at Table 4.7, it is easy to notice how PPO can still find allocations of service components that achieve an objective function score close to 4, without retraining the model. This seems to confirm the good performance of PPO for the service management problem discussed. Moreover, PPO can achieve this result after an average of 36 steps, i.e., each one corresponding to an evaluation of the objective function. This was the result of the longer training procedures that on-policy DRL algorithms require. On the other hand, as is visible in Table 4.7, the DQN showed a strong performance degradation, as the best solutions found during the 30 test episodes had an average of 1.60. Therefore, the DQN demonstrated lower adaptability when compared to PPO in solving the problem discussed. With regard to the CI algorithms, GA was the worst in terms of the average values of the objective functions, while all the other algorithms achieved average scores higher than 4.0. As for the average number of infeasible allocations, the constrained versions achieved remarkable results, especially QPSO-ECT and MPSO-ECT, where the number of infeasible allocations was zero or close to zero for GWO-ECT. Differently, the GA-ECT and PSO-ECT could not minimize the number of infeasible allocations to zero. Overall, MPSO was the algorithm that achieved the highest score at the cost of a higher number of iterations. From a sample efficiency perspective, Table 4.7 shows that QPSO was the CI algorithm that achieved the best result in terms of the number of evaluations of the objective function. At the same time, MPSO-ECT showed that it found its best solution with an average of 1493 steps, which was considerably lower than the 7432 steps required by MPSO, which, in turn, found the average best solution overall. Finally, GA and GA-ECT were the CI algorithms that required a larger number of steps to find their best solutions.

Furthermore, Tables 4.7, 4.8, 4.9, and 4.10 show the reasons for the poor performance of the DQN: four out of six services had a PSR less than 50%. More specifically, the PSR for the safety service was 0%, and the one for healthcare was 5%. Even PPO was not great in terms of the PSR in the what-if scenario. On the other hand, all CI algorithms except for GA and GA-ECT recorded PSR values above 90% for all services, thus demonstrating that the cold restart technique was effective in exploring optimal solutions in the modified search space. Overall, PPO was demonstrated to be effective in exploiting the experience built on the previous training even in the modified computing scenario. Contrarily, the DQN did not seem to be as effective as PPO in reallocating services' instances in the what-if experiment. On the other hand, the training of CI algorithms does not create a knowledge base that these algorithms can exploit when the scenario changes remarkably. However, the cold restart technique was effective in re-optimizing the allocation of service instances.

4.2 The Value-of-Information Connection

The proliferation of IoT sensors and devices is generating a deluge of data which is often processed in cloud computing facilities far away from where the IoT data were generated. Despite new generation communication technologies, i.e., 5G and beyond, promise higher bandwidth and reliability, distributing processing along the CC represents a promising approach in terms of lower service latency and communication overhead [165]. This suggests the opportunity to experiment with information-centric approaches [166] and lossy and adaptive service models [167] that focus on the processing and dissemination of important data. In this regards, VoI represents a compelling approach to rank information and services according to the actual value they provide to end users [168], [169]. In fact, VoI-based approaches enable to maximize resource utilization from a user perspective in a straightforward fashion across the entire CC, also addressing the issue of resource sharing among competing services.

Orchestration tools so far have focused on providing a seamless deployment solution for containerized microservices. Despite remarkable progress in recent years, also in CC environments and applications, they have relatively neglected the opportunity to integrate with service components at the QoS tuning level [170]. Instead, CC environments call for new resource management strategies with a holistic perspective of the whole computing ecosystem, that consider opportunities in both resource reallocation - either horizontally (across nodes on the same cloud, fog, or edge layer) or vertically (across multiple layers) - and at the service QoS reconfiguration levels [171]. Therefore, task allocation/re-modulation processes, when a specific microservice requires more processing capabilities or responding to new QoS requirements, need to consider whether to expand workloads horizontally, i.e., by allocating more or different resources to the software component in the same computing level (e.g., at the edge), or vertically, i.e., by migrating the software component to a node with a bigger computational capacity or to upper layers (e.g., from the fog to the cloud).

At the same time, there is the need to address the gap between an ever-increasing deluge of information and the (relative) resource scarcity at the edge, where this information is produced and often consumed. In fact, the massive introduction of IoT devices and the increasing need for computation closer to users, to avoid processing latencies caused by the transfer and processing of data in the cloud, are arguably the main reasons why CC nowadays has a significant and growing interest.

These challenges introduced by CC environments suggest the opportunity to push forward the state of the art and experiment with lossy and adaptive service models. In this context, VoI methodologies and tools represent an innovative and interesting foundation for the realization of immersive and adaptive services. *VoI is a subjective measure that allows quantifying the value that an information provides to its users*. VoI-based methods explicitly consider the different characteristics of Information Objects (IOs), tracking the value/utility of IOs during their processing and dissemination phases, thus

enabling to rank these IOs according to their importance and relevance and to prioritize the processing and dissemination of the most important IOs, while discarding low value ones [167]. In turn, ranking services and microservice components according to the total amount of VoI they provide to end users represents a natural and effective approach to realize self-adaptive services for fog computing applications.

Using the amount of VoI delivered to end users as a resource assignment criterion, a system will be able to naturally and seamlessly prioritize the assignment of resources to microservices that provide the highest value to their end users, either because they serve a considerable amount of users or because they provide highly valuable information [169]. Factors such as the availability of resources or the changing demands of applications could be taken into account to estimate the relevance of requests and dynamically allocate resources in the CC environment. Thus, the modulation of VoI-based message processing policies allows decision-making components to naturally adapt to current resource availability and computing context. Therefore, in case of resource abundance and high VoI delivered to end users, every generated message will ideally be processed by these components. However, in case of resource scarcity and/or an abundance of competing services that try to compete for the same resources, decision-making components will prioritize the processing and delivery of messages with a higher VoI and discard those with a lower VoI.

4.2.1 The VOICE platform

To enable resource allocation all along the CC while delivering the highest possible value to end users, this subsection presents *VOICE*[2], an innovative platform that allows the run and management of services with dynamic topologies on the CC [172], with service components running on devices exhibiting highly heterogeneous capabilities, in terms of storage, connectivity, and performance. VOICE aims to push forward the state of the art for service elasticity management in the CC by explicitly involving application components in expansion or contraction operations of resource allocation, to ensure the best QoS levels according to conditions and application runtime requirements. VOICE builds on top of previous research on adaptive VoI-based service models and runtimes [169], CI-based optimization solutions [106], and innovative DT approaches [173], [174] to provide a full featured orchestration solution designed for the CC.

VOICE conceives the CC as an ensemble of heterogeneous computing clusters to enable the development, dynamic deployment, and management of elastic IT services within and across multiple and heterogeneous clusters, each containing different types of nodes (i.e., L1: edge, L2: fog, L3: cloud), as depicted in Figure 4.3. Different types of

[2]M. Zaccarini, B. Cantelli, M. Fazio, et al., "Voice: Value-of-information for compute continuum ecosystems," in 2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN), 2024, pp. 73–80. doi: 10.1109/ICIN60470. 2024.10494500

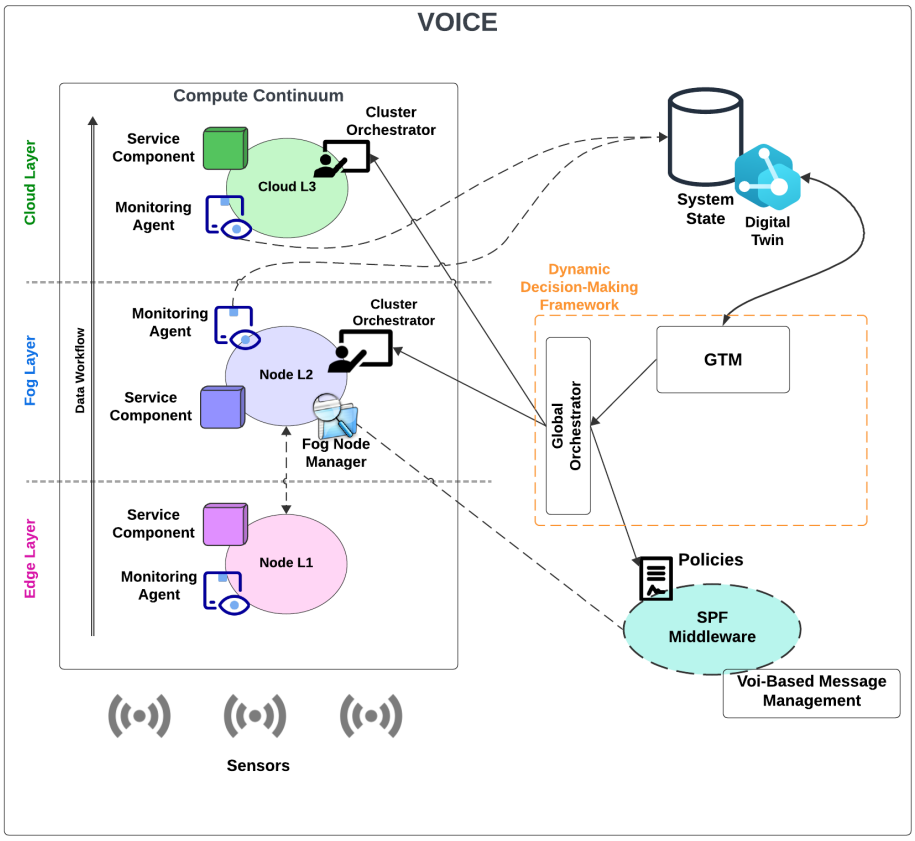


Figure 4.3: Global picture of the VOICE architecture

nodes in the system give different opportunities for data processing and software execution. Specifically, VOICE services are realized through the composition of clearly separated but interconnected components (i.e. microservices) that can be independently deployed and executed. Each service has a description document that identifies the requirements that each component has and the connections between components. Our main purpose is to tackle the challenge of combining various execution platforms, from smart IoT devices, useful to collect data from the environment and trigger events, to geographically distributed edge nodes and from fixed or mobile fog nodes to the Cloud, into an ubiquitous and seamless execution environment. However, their interoperability represents a challenge due to the adoption of different technologies at different levels and the different perspectives in the management of resources.

Furthermore, VOICE is able to manage all the available resources in the CC implementing a dynamic and multi-layer orchestration solution with differentiated strategies to optimize service deployments and scalability at different levels. This is the key approach for a seamless and pervasive computing system able to optimize performance of

running applications and services. VOICE adopts "Sieve, Process, and Forward" (SPF) as a reference service component execution runtime supporting VoI-based processing [167]. SPF provides a powerful set of concepts and tools for the development of VoI-based service fabrics. VOICE exploit SPF to build a multi-layer resource management framework that extends state-of-the-art VoI modeling to define specific VoI evaluation solutions tailored to the type of information to be processed.

VOICE introduces a multi-layer orchestration solution to overcome specific resource management needs at different levels, with a dedicated resource management layer built on top of the Global Orchestrator, Cluster Orchestrator, and Fog Node Manager components. The Global Orchestrator is in charge of configuring and coordinating the deployment of applications and services across computing clusters and nodes at different levels, elaborating information from monitoring agents and enabling the CC with specific mechanisms that allow software components to move vertically along the resource level stack. Each fog node connected to the VOICE platform has a Fog Node Manager component, which is in charge of abstracting the properties of the specific physical/virtual node and dynamically allocating a given share of available resources to one or more microservice components according to the requests of the Global Orchestrator. The Node Manager will implement potential enhancements to the computing resource management at a local level working on the reservation of computing resources, the placement of the virtual elements contributing to a service, the performance efficiency, and the mutual isolation of virtualized workloads sharing the same hardware substrate.

To optimize service deployments and resource allocation along the CC, VOICE develops a dedicated dynamic decision-making framework. GTM is the software entity responsible for driving the allocation of the service components to the nodes at different levels to meet their demand for computational resources. Such decisions are then passed to the Global Orchestrator, which is in charge of carrying out the actual deployment operations. To perform its duties, GTM has perfect knowledge of the computational capacity of each device as well as the demand of each service component and their mutual interactions, beside the service priority and expected QoS. The aims are manifold: first, the demand of each service component should be fully satisfied, meaning that the number of service components allocated to the cloud should be minimized while keeping latency-sensitive service components at the edge. Second, service components with a high level of interaction should be allocated on the same node to minimize overall latency. Third, VOICE would try to allocate service components to maximize the VoI delivered to their users. At the Edge level, the Cluster Orchestrator is a distributed software entity that automates the deployment of service topologies in a cluster of edge nodes, allowing resource reallocation for horizontal scalability.

Furthermore, VOICE features a distributed monitoring subsystem that not only allows to collect standard performance metrics such as response times, etc., but also implements the monitoring of VoI generation at both the single service component and

at the entire service level. This information allows the VOICE resource management subsystem to make well-informed and cognitive service instantiation and adaptation decisions. The distributed monitoring subsystem of the VOICE platform leverages a capillary infrastructure of dedicated monitoring agent components. Furthermore, differentiated versions of the monitoring agents along the CC provide monitoring capabilities for the global computing infrastructure and QoS of running applications. To do so, monitoring agents will monitor both the availability of node resources and the current state of the service components running in their control domain.

CI solutions allow the VOICE orchestration components to execute the monitoring policies in a very reactive way, overcoming the problem typical of traditional monitoring strategies that can only provide the current information on node resources and services performance, limiting the vision of the logic behind the dynamic variation of system behavior. However, we designed VOICE to realize proactive service reconfigurations by taking advantage of a DT approach, that allows to create a virtual replica of a real system, i.e., the CC infrastructure and services, for what-if scenario analysis and system optimization purposes [51], [175]. The VOICE GTM can leverage the DT of the system to evaluate the performance of a service in an alternative deployment and configuration, thus avoiding harm to the configuration of the real-world system. The DT will also provide an effective platform for experimentation with a wide range of VoI policies, with the objective of identifying the most promising ones. The VOICE DT can be configured to use different subsystems to recreate a complex IT service in the CC. The default option is Phileas, already described briefly in the previous section. Phileas features a relatively sophisticated service model that accurately evaluates the VoI produced by each single service component across the entire service fabric. As a result, it represents a very good fit for VOICE.

The VOICE GTM goes beyond the current state-of-the-art by introducing scheduling strategies that consider both heterogeneous resource availability in different nodes and the total VoI optimization criterion for resource assignment. Different examples of them are provided in the next subsection, where experiments show how VOICE is able to adapt the distribution of allocated resources depending on its final objective. Toward that goal, GTM adopts sophisticated CI solutions, such as GA and QPSO, that can quickly explore the very large and complex space of feasible IT service configuration to identify the optimal one - also leveraging a DT. Along with other CI-based solutions, both GA and QPSO have demonstrated remarkable flexibility and performance in managing resources in CC scenarios [106], especially due to their ease of implementation and ability to avoid stagnation in local minima due to their stochastic nature.

4.2.2 VOICE Use Case and Evaluation

As a reference scenario for the adoption of the VOICE platform, we propose a smart city scenario characterized by several service components that must be allocated on the

CC in a way that maximizes the total VoI delivered to end-users. We suppose that the smart city provides a multitude of cameras, which collect video feeds from the surroundings to feed a plethora of smart city services such as traffic monitoring, object tracking, and object detection applications. We defined a total of six different services: pollution, traffic, audio, video, healthcare, and safety. The pollution service estimates the air quality in several locations in the smart city and notifies citizens via mobile devices if they have a high exposure to polluted air. The traffic service collects information about the traffic flows in the urban environment to let citizens know what routes to take, e.g., to avoid congestion and heavily polluted areas. On the other hand, the audio service provides analysis based on acoustic data, e.g. noise pollution caused by traffic, construction sites, industries, and so on. The video service instead analyzes camera feeds from nearby CCTVs and other video content, e.g., social media, to run machine learning processing. Healthcare is a compelling topic often associated with smart city environments, mobile telecommunications, and IoT. Therefore, we describe a generic healthcare service for this use case. Finally, the safety service provides a variety of services, including social safety, antiterrorism, and crime prevention efforts. As for video services, the latter two are defined as time-sensitive and more critical than traffic, pollution, and audio services. In fact, there are many cases where an immediate response or action is required, such as locating a potential threat or finding a missing person.

Several instances of these services would be deployed on top of 13 computing servers distributed in the cloud, fog, and edge levels. Specifically, for these experiments, we assume that 10 servers would be available at the edge level, and 2 at the fog level, and we model a single cloud entity with unlimited resources. This can be a representative example of a CC scenario where the majority of the service components would exploit the computing capabilities in close proximity of users and sensors. We also imagined that there would be several citizens who would interact with these services. Therefore, we specify 10 different user groups that generate services' requests. Both devices and users are located in 13 different locations, all characterized by latitude and longitude coordinates according to the GPS and with starting VoI values. In addition, for each type of service, we described two different decay functions, a linear decay used for pollution, traffic, and audio services, and an exponential decay which is used for time-sensitive services. Data sources associated with time-sensitive services would generate requests with a higher starting VoI value, as we assume these services as more significant and urgent from users perspective. For the following experiments, we modeled the VoI values in a $[0, 1]$ interval, where 0 corresponds to the lowest VoI possible and 1 to the highest. The VoI associated with time-sensitive services would have a stronger time decay compared to standalone / batch services. It is therefore important to process time-sensitive requests as quickly as possible to minimize the decay information objects are subject to from their origination to their processing and delivery.

With regard to communication modeling, to approximate communication latency between the different levels of the CC, the measurements available on the CloudPing

Table 4.11: Service Metrics used in Experimental Evaluation

Service Type	Time Decay	Data Sources	Avg. Message Generation Moderate Load (Msg/Sec)	Avg. Message Generation Heavy Load (Msg/Sec)	Starting VoI	
					Mean	Standard Deviation
Audio	Linear	1	29	100	0.4	0.1
Healthcare	Exponential	2	9	9	0.8	0.05
Pollution	Linear	1	40	125	0.4	0.1
Safety	Exponential	2	8	8	0.8	0.05
Traffic	Linear	1	40	100	0.4	0.1
Video	Exponential	3	10	10	0.8	0.05

website [176] were considered as a baseline, from which three pairs of locations whose communication latency could be a reasonable approximation of a computing scenario composed of connected edge, fog, and cloud resources have been selected. For each pair of locations, the parameters for modeling a random variable with a truncated Gaussian distribution were calculated. The same type of variable is adopted to model the starting VoI value for each type of service in the scenario. During simulation, each time a new request of a specific type is generated, the starting VoI is computed according to its specific mean and standard deviation configuration, as presented in Table 4.11. Specifically, there are two main service classes: low VoI services, which include Audio, Pollution, and Traffic, and high VoI services, which include Healthcare, Safety, and Video. Each of these classes has its own characteristics, such as time decay type (Linear for low VoI services and Exponential for high VoI service), number of respective data sources, and amount of requests per second depending on the load scenario. All these metrics are visible in Table 4.11. Furthermore, the longer each request waits in the relative queue, the more its VoI decays and the less likely the available resources will be allocated for its processing. Ideally, if a request remains stuck in the waiting queue for too long, its VoI decreases remarkably, and users may no longer be strongly interested in receiving the computation results. Finally, considering computing modeling, we assume that each service instance processes the incoming requests sequentially in a queue fashion. More specifically, we model the execution time of each service component with the adoption of a random variable with exponential distribution characterized by a specified parameter.

The VOICE DT component reenacted this use case, leveraging the Phileas sim-

ulator, to compare three different optimization criteria for service orchestration in the CC: VoI maximization, latency minimization, and request completion ratio maximization, i.e., the number of requests processed within the simulation time window. These methodologies have been tested in two different scenarios: the first is a moderate load scenario, characterized by nearly 10,000 requests. Instead, the second is a heavy load scenario for a total of more than 20,000 total requests, with a remarkable increase of low VoI requests compared to the first outline. In a second experimental phase, the exploitation of VoI took a further step. By manipulating a VoI defined for each service, CI approaches demonstrated their ability to find the configuration that best exploits the scarcity of resources to compute high-value requests, with the aim of overcoming the limitations observed in the heavy load scenario during the first evaluation.

The various criteria have been compared by implementing a GA and a QPSO based optimizers and by defining three fitness functions corresponding to the optimization criteria. For all experiments, both GA and QPSO optimizers ran for a total of 250 generations. Each generation corresponds to 128 and 40 evaluations, i.e. a simulation run, of the fitness function, respectively. Each simulation has a one-minute duration (including 10 seconds of warm-up), corresponding to the generation of roughly 10,000 service requests in the moderate load scenario and more than 20,000 in the heavy load one. This is possible by manipulating the exponential random variables that model the time between the generation of subsequent service requests. The orchestrator determines where to allocate instances for the defined service components among the CC devices. The service orchestrator can also activate multiple instances of the same service component on different devices to distribute the application load and improve performance when necessary. Finally, a computing device can run multiple instances of different service components, according to its computing capabilities. For an initial experimentation run, the VoI thresholds were excluded to make a comparison between strategies as fair as possible for both scenarios.

For each optimization criterion, the best configuration of the service component was selected for both load scenarios, in terms of the fitness function. Figure 4.4 reports the results, showing the total VoI, the request completion ratio, and the ratio between this ratio and the average latency (measured as the sum of the transfer time of requests) for all three configurations of both GA and QPSO approaches. As expected, in both scenarios the VoI approach reports the highest VoI value. At the same time, it achieves a very competitive request completion rate. Regarding its PR/Latency outcome, it remains consistent with the results obtained by the "Max Ratio" approach. This is expected, as both methodologies aim to process as many requests as possible without considering other optimization aspects. This can easily lead to a higher average latency as a consequence, reducing the PR/latency value. Moreover, to evaluate the validity of the VoI approach, it is important to examine its component allocation efficiency at every CC layer compared to other strategies. Table 4.12 illustrates how the different methodologies place multiple service components instances. The results pro-

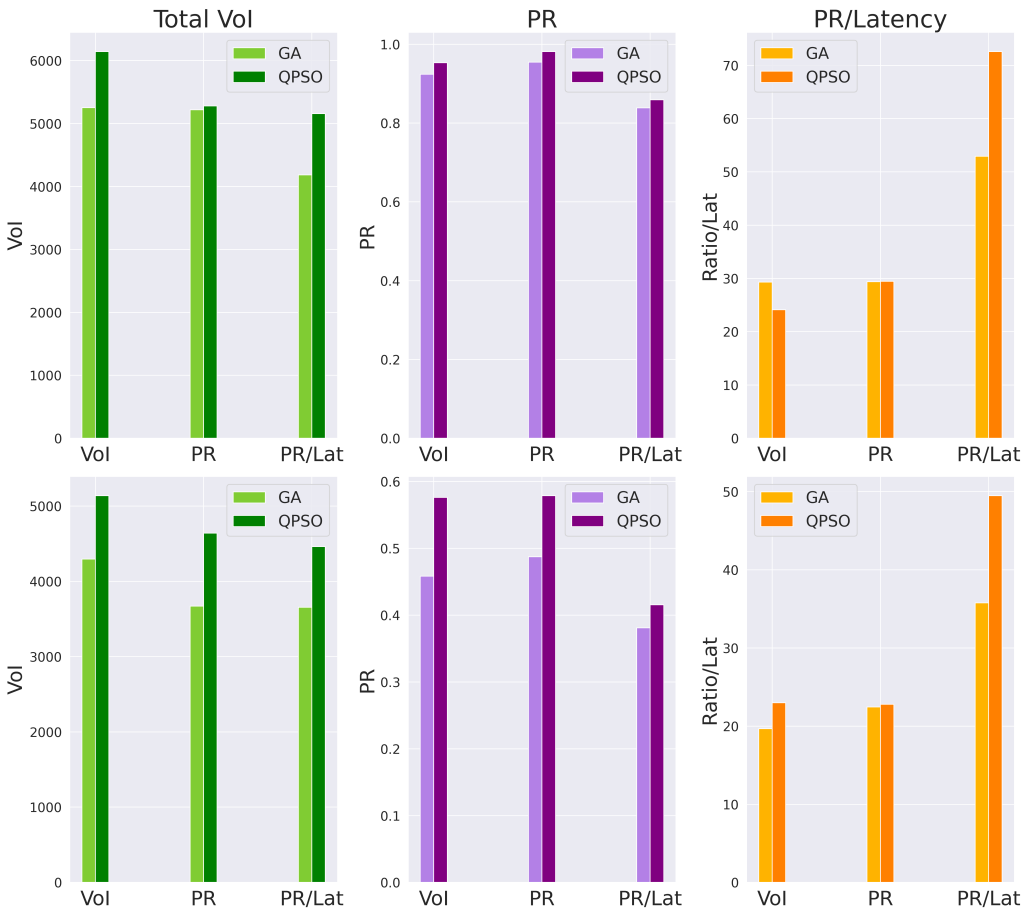


Figure 4.4: Results of the three different optimization methodologies using GA and QPSO as optimization algorithms.

vided once again prove the effectiveness of the VoI-based approach. In fact, it is capable of obtaining very similar performance compared to the "Max Ratio" strategy for every load, but with resource savings up to 10% in the GA case and 30% for the QPSO. In the last configuration, it is evident that it achieves the lowest value of total requests processed and generated VoI. As mentioned above, this is an understandable behavior, since the system cannot rely on the allocation of numerous service components (especially on the fog and the cloud layer) to minimize the latency. In fact, the more services components are allocated, the more cumulative latency increases, causing a deterioration of the mean latency and, consequentially, of the final PR/Latency rate. This is even more evident in both tables, which show that the Ratio/Latency configuration is the one that allocates the fewest service components in every occurrence.

In a second experimental phase, both optimization approaches must handle a new parameter for each type of service component: a VoI-based threshold to filter requests

Table 4.12: Service components allocated

Moderate Load					
Configuration	**Metaheuristic**	**Edge**	**Fog**	**Cloud**	**Total**
Max VoI	GA	37	6	4	47
Max Ratio	GA	41	8	5	54
Ratio/Latency	GA	**34**	3	**0**	**37**
Max VoI	QPSO	40	2	6	48
Max Ratio	QPSO	55	10	6	71
Ratio/Latency	QPSO	43	**0**	**0**	43
Heavy Load					
Configuration	**Metaheuristic**	**Edge**	**Fog**	**Cloud**	**Total**
Max VoI	GA	**37**	3	5	45
Max Ratio	GA	38	11	2	51
Ratio/Latency	GA	**37**	3	1	41
Max VoI	QPSO	53	6	6	65
Max Ratio	QPSO	59	12	6	77
Ratio/Latency	QPSO	39	**0**	**0**	**39**

still considered valuable from those for which it is not worth allocating resources. By manipulating these thresholds, both GA and QPSO demonstrate their ability to identify the configuration that best exploits the scarcity of available resources. This prioritizes the processing of high-value requests for the end user. To employ filtering capabilities and overcome the shortcomings exposed in the previous evaluation, service thresholds were applied in the scenario defined as heavy load. Specifically, both orchestrators were run for a total of 500 generations each with the same number of evaluations of the previous evaluation. This enabled them to better explore and find a more effective solution in this very dynamic proposed scenario. During the process, orchestrators could choose between 4 possible threshold values: 0.0, 0.125, 0.250, and 0.5. Once set, these values remained constant until the next simulation. Figures 4.5 and 4.6 illustrate the evolution of the total VoI obtained in relation to the varying thresholds throughout the optimization process conducted respectively by GA and QPSO. Specifically, the GA-based optimizer reaches a very similar VoI value to the previous in the heavy load scenario (visible in the bottom-left plot in Figure 4.4). However, in this case, it achieves this while processing over 3,000 fewer requests than before. This indicates that the thresholds effectively filtered out most low VoI requests, allowing system resources to be reserved for processing requests with high VoI value. In contrast, QPSO performed significantly better. Indeed, thanks to its threshold manipulation, it identi-

Table 4.13: Service components allocated with threshold filtering

Configuration	Metaheuristic	Edge	Fog	Cloud	Total
VoI Thresholds	GA	**40**	**5**	**5**	**50**
VoI Thresholds	QPSO	44	6	6	56

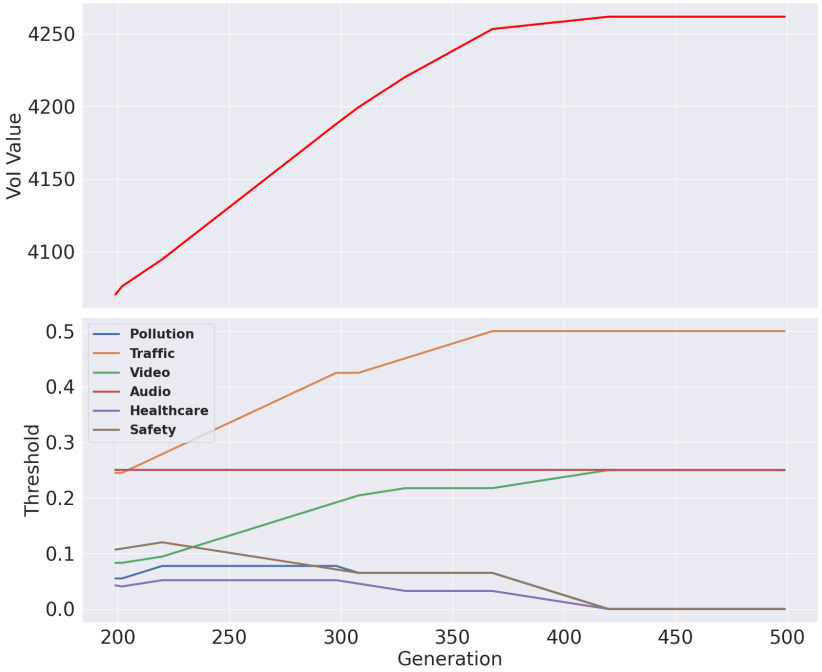


Figure 4.5: Total VoI evolution in relation to the service thresholds manipulation by GA.

fied a solution that achieves a better VoI with several thousand fewer processed requests along with a notable resources aving (56 total components allocated as visible in Table 4.13 compared to 65 in Table 4.12). In both cases, due to the remarkable increase of low VoI service requests in the heavy load scenario, their relative thresholds are predominant. Specifically, the QPSO optimizer achieves the best VoI values when it sets all low VoI service thresholds above the others (0.25 regarding the Traffic service and 0.125 for both Audio and Pollution). However, while GA can improve its efficiency, it is not as effective as QPSO. In fact, GA tends to increase the threshold for the video service component instead of the Pollution one. Due to the fact that Pollution is not considered as critical as Video, this leads to a service component configuration that delivered a lower amount of VoI than the one generated with QPSO.

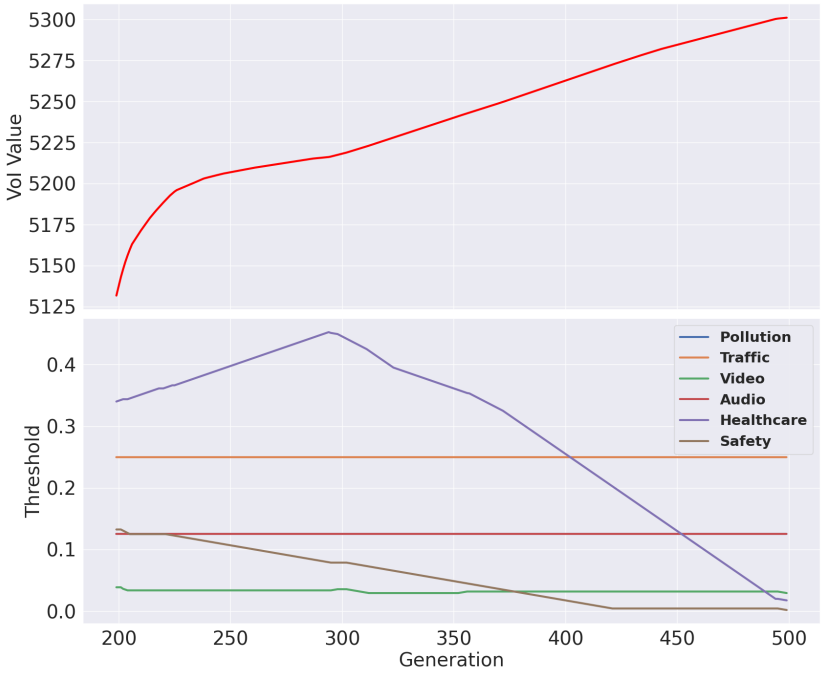


Figure 4.6: Total VoI evolution in relation to the service thresholds manipulation by QPSO.

4.3 Hybridized Hot Restart via Reinforcement Learning for Compute Continuum Orchestration

As previously highlighted, service and network management in the CC remains a compelling challenge for both academia and industry. Applications deployed in these environments must adhere to strict Key Performance Indicators, particularly regarding latency and resource efficiency. This requires careful strategies for placement, migration, and resource allocation to ensure that the system can dynamically adapt to variations in workload and network conditions. RL and CI techniques have been proposed by the research community to address microservice deployment and resource management challenges at various layers of the CC. The firsts have shown promising results because of their ability to learn optimal strategies through continuous interactions with the environment. Alternatively, metaheuristics offer an advantage in solving complex optimization problems more efficiently, often reducing computational time compared to traditional mathematical optimization techniques. The application of RL and CI techniques has been further facilitated by the introduction of the DT paradigm also for CC ecosystems [51]. Tools such as KT and Dantalion [177] enable what-if scenario analysis on the virtual element of DT to evaluate multiple deployment configurations

in a faster and more efficient way. However, the dynamic nature of CC environments calls for continuous re-optimization, e.g., node disconnections or workload variations might happen frequently, thus requiring even more efficient optimization solutions. Based on the work presented previously, this section aims to discuss the combination of the strengths of both RL and CI to realize more efficient optimization solutions[3]. Section 4.1 showed that CI techniques can efficiently explore the search space and find near-optimal solutions, but tend to struggle in cases of mild and severe system changes. Differently, RL techniques, particularly PPO, demonstrate remarkable adaptability to dynamic scenarios without requiring retraining. This hybridization technique leverages RL as a tool to enable hot restart of CI techniques. Rather than retaining only part of the memory acquired during the exploration process, RL can exploit their knowledge base to provide an advantageous starting point for restarting the CI techniques in a promising region of the search space.

In fact, although RL methods tend to be more resilient in dynamic scenarios and can generate new configurations more quickly than CI tools, these configurations can be further improved by performing a metaheuristic. Specifically, leveraging the stronger resilience capabilities of RL to generate a starting solution, the metaheuristic can focus its search on the portion of the search space suggested by RL, thereby requiring fewer evaluations of the fitness function. With this regard, cold and hot restart techniques are two distinct approaches for re-optimizing a metaheuristic when its target function changes. In a cold restart, a metaheuristic restarts without any memory of the previous system optimization. In contrast, a hot restart refers to a new optimization process that retains entirely or partially the memory of the previous one. Using the latter, a metaheuristic should converge more quickly to the new profitable configuration of the system. However, the effectiveness of such techniques depends on several factors, especially the distance of the new objective from the previous one. If the system undergoes a significant change, the new optimum could be far from the one the metaheuristic remembers from its accumulated experience. For this reason, this investigation proposes different hot restart techniques that, instead of relying entirely on metaheuristic memory, leverage RL to create a "new memory" that can be used entirely or as part of the starting point of the optimization process in the novel scenario, as illustrated in Figure 4.7. The idea is that an RL agent should be more capable of adapting its decisions according to system changes and guide the agent toward a state that maximizes the reward, having already experienced different scenarios during training with the help of DT for their generation. This new state then serves as the initial population for metaheuristics, which can be defined by relying entirely or partially on the solution originated by the RL approach. For example, it can be combined with elements from metaheuristic

[3]M. Zaccarini, F. Poltronieri, C. Stefanelli, and M. Tortonesi, "Hybridized hot restart via reinforcement learning for microservice orchestration," in NOMS 2025-2025 IEEE Network Operations and Management Symposium, 2025, pp. 1–7. doi: 10.1109/NOMS57970.2025.11073715.

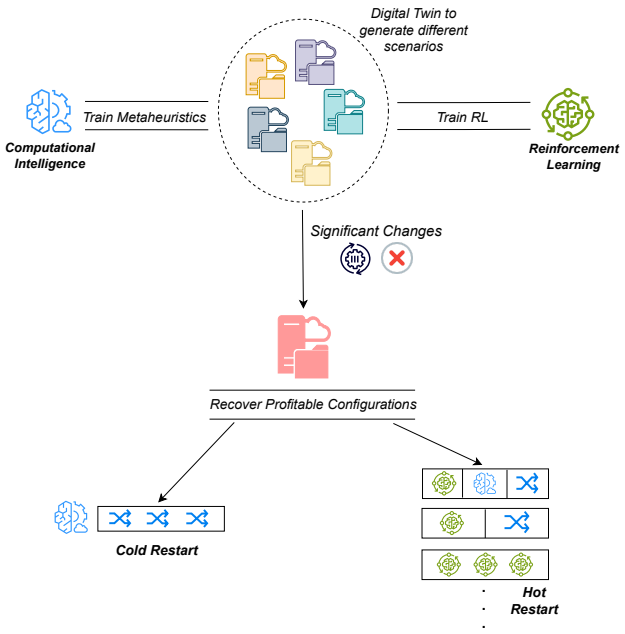


Figure 4.7: Exploitation of cold and hot restart approaches.

memory, generated randomly starting its search from a promising point identified by RL, allowing it to explore a wider search space and reducing the probability of getting stuck in local optima or restricted regions.

The procedure is executed as follows:

1. Firstly, the procedure initializes the population randomly to allow the metaheuristic to explore the search space in its entirety, by improving the potential solution found at each step until the termination criterion is met, e.g. a fixed number of iterations.

2. If changes in the scenario are detected, the procedure restarts the exploration of the search space to find another optimized solution.

3. To do so, it can select between different variants of a hybrid hot restart that keeps part of the elite members of the previous population and combines this population with an RL-generated solution and a more traditional cold restart technique that reinitializes the entire population.

To implement the hybrid hot restart technique, each CI algorithm is trained and tested in a baseline scenario with stable conditions regarding both available resources and generated traffic. When such significant changes occur, optimization strategies

should be able to adapt quickly and maintain a satisfactory Quality of Experience for users, despite changes in the ecosystem. Therefore, to accelerate re-optimization, the proposal is to run the trained RL to find a new state and then start a new optimization process with a reduced number of iterations, as the algorithm should begin its exploration in a promising area. To validate the hybrid hot and cold restart approaches, the same objective function 4.7 formulated in Section 4.1 has been considered. Specifically, the function $f(x)$ measures the output during a simulation window as the PSR for each application, weighted by an importance factor θ, which value is defined by the criticality assigned to each microservice in the scenario. Φ represents the number of infeasible allocations, which occur when an instance of a service component is allocated to a computing node that lacks sufficient CPU or memory resources. The goal of this work is to find solutions that maximize the number of satisfied requests while respecting the resource constraints imposed by the CC. Therefore, the metaheuristics used are "Enforced ConstrainT" versions called respectively GA-ECT, PSO-ECT, and GWO-ECT, as defined previously. These adapted metaheuristics incorporate a penalty factor (Equation 4.6) related to the number of infeasible allocations, which acts as a gradient during the optimization process to maximize the objective function while minimizing impractical solutions. It is important to note that the implemented system model can deal with infeasible allocations by not allocating instances to nodes when there are not sufficient resources available. This is similar to deploying an excessive number of replicas in an overloaded K8s cluster, i.e., when resources are depleted, replicas will remain pending. However, even if infeasible allocations are accepted, it is important that the metaheuristics can provide feasible solutions to the problem. The libraries considered are analogous to those discussed in Section 4.1 (ruby-mhl for metaheuristics and Stable Baselines3 for RL methodologies, respectively).

Also here, the Phileas simulator was leveraged to reenact the behavior and the dynamics of a smart-city environment where microservices are deployed at various layers. Regarding the computing resources available across the CC, virtually infinite computing resources available for rental in cloud data centers were modeled. In the fog layer, the scenario included two devices with 50 units of resources for each one of them. Finally, 10 devices with 25 units per device were placed in the edge layer. After a certain period, a sudden shutdown of the cloud layer was simulated, mimicking CE methodologies similar to Chaos Gorilla designed by Netflix [178] to shift the new optimum of the system through a strong stress test.

To make possible the procedure illustrated before, a trained RL agent capable of generating allocation solutions is required. For this evaluation, the selected PPO implementation was trained for 50.000 steps in the baseline scenario, i.e., the one where all the computing layers were working properly. Then, we decided to evaluate four different population initialization strategies for the hybrid hot restart. The first strategy aimed to exploit the memory of the previous optimization by injecting 60% of the fittest members into the new population, along with copies of the best solution generated by

Table 4.14: Evolutionary Algorithms characteristics between cold and hot restart.

Evolutionary Algorithm	Cold Restart		Hot Restart	
	Population Size	Iterations	Population Size	Iterations
GA	128	50	128	30
PSO	40	50	40	30
GWO	30	50	30	30

the PPO agent. The second approach used 40% of the fittest members of the old population and initialized the remaining individuals with the best solution generated by PPO. The third strategy introduced a random component that generated 30% of the new population with random individuals, while 40% came from the old population and the final 30% from the best solution found by the PPO agent. Finally, the last strategy initialized the new population entirely with randomized elements based on the best solution found by PPO. Specifically, each element was randomly increased by one, decreased by one, or left unchanged. These variations define a region of the search space around a promising RL-derived solution, helping the metaheuristic to quickly converge to a profitable configuration. On the other hand, when adopting a cold restart, metaheuristics maintained their usual behavior, i.e., the new population is entirely randomly generated. By not considering the memory of the previous optimization, the cold approach allows metaheuristics to explore a larger region of the new search space. However, it could be more difficult to find new local optima in the same number of iterations.

20 optimization runs have been performed for each metaheuristic to collect statistically significant results, capturing both the differences at the algorithm and strategy levels. Each run comprised 80 iterations divided into two different phases: in the first phase, each approach starts from a random allocation and performs an optimization process 50 iterations long, aiming to find the best service component allocation to handle the load generated by users. Next, a sudden shutdown of the cloud layer was simulated, requiring all approaches to quickly adapt their solutions and maintain high Quality-of-Experience for end-users, despite the significant failure suffered by the environment. Each metaheuristic then continued for 30 more iterations in both cold and hot restart versions, trying to restore acceptable service conditions across the remaining layers of the CC. Table 4.14 reports the configurations for each metaheuristic. GA was configured to use a population of 128 elements, PSO with a swarm size of 40 particles, and GWO to use a swarm of 30 wolves.

Figures 4.8, 4.9, and 4.10 provide a visualization of 20 optimization traces for the algorithms studied. Specifically, they show the progress of the two optimization stages,

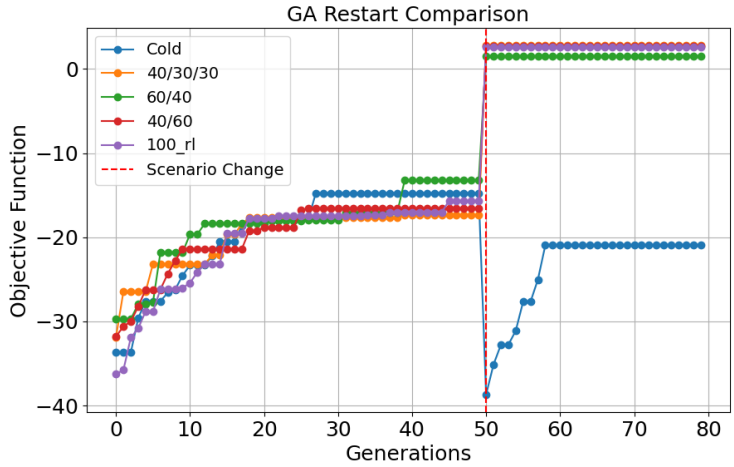


Figure 4.8: Exploitation of cold and hot restart approaches for GA algorithm.

separated by a red vertical line, using the four hot restart strategies and the cold restart. Using hot restart, all metaheuristics can find solutions with values similar to, and for PSO even higher, the one found in the baseline scenario. In contrast, cold restart performs worse in all settings, demonstrating to be far less convenient. From an algorithm perspective, PSO and GWO outperform GA, with PSO showing a steady improvement beyond the 50th iteration, suggesting strong exploration capabilities and robustness to significant scenario changes when exploited with the hot restart. In contrast, GWO initially converges very quickly, but is more affected by the deactivation of the cloud layer. However, it is capable of recovering the previous operating conditions within the 30 generations available using each hot restart strategy. Furthermore, it is also interesting to note how its cold restart configuration is able to reach a result that is similar to the other restart strategies. As a consequence, probably the complexity and the computational load introduced by a PPO-based hot restart may not be strictly required in the GWO adaptation process thanks to its capabilities in exploring efficiently the search space, even in dynamic scenarios like the one considered.

More statistically significant results are visible in Table 4.15, which illustrates the mean value and standard deviation of the objective function across the 20 different optimization runs of each approach. Table 4.15 shows how significant is the contribution of PPO in the generation of parts of the new population (i.e. 60%, 40%, 100%) since each combination of RL-generated elements and previous fittest members improves the performance of all metaheuristics when the scenario changes. The reason behind this is that the hot restart approach was able to spot an advantageous starting point in terms of minimizing the number of infeasible allocations, minimizing its impact on $f(x)$ as defined in Equation 4.7. In particular, there are no significant differences among the various combinations of RL and CI strategies. This seems to indicate that PPO brings

Figure 4.9: Exploitation of cold and hot restart approaches for PSO algorithm.

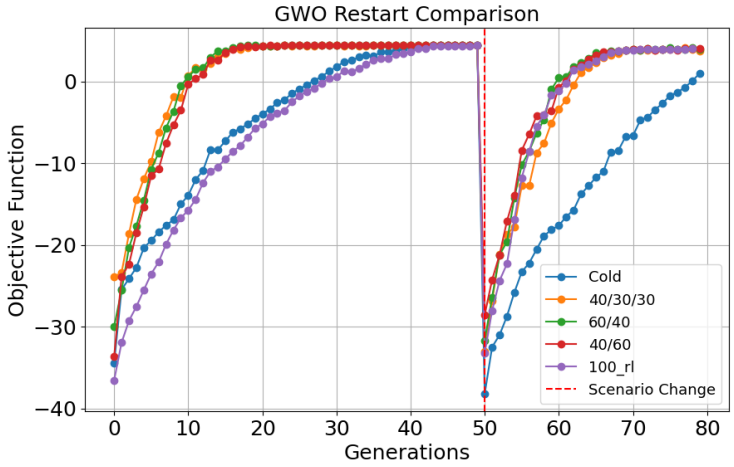


Figure 4.10: Exploitation of cold and hot restart approaches for GWO algorithm.

the highest contributions in the process. Looking at each algorithm singularly in Table 4.15, GA obtained the lowest in terms of the objective function metric. At the same time, it was not able to improve the starting position provided by RL in many optimization runs. Probably, with the configuration tested, GA was not very effective in exploring the space due to the limited area where the new population was gathered. Considering GWO, all hot restart strategies allow the algorithm to converge to values considerably higher than the average of 0.13099 achieved using the cold restart, earning the best objective function for each hot restart strategy considered. Furthermore, GWO works well even in a cold restart configuration when compared to GA and PSO. This

demonstrates a quick convergence ability, especially considering that at each iteration GWO evaluates only 30 individuals instead of the 128 of GA and the 40 of PSO. Finally, the hot restart allows PSO to explore a much more profitable area of the search space and to improve the solution considerably compared to a cold restart. In conclusion, hot restart demonstrated important capabilities to assist metaheuristic approaches, making it an effective solution for quickly recovering from sudden and potentially disruptive changes in CC ecosystems similar to the one under consideration.

Considering that these results do not indicate significant differences between the different strategies, an additional experimental phase analyzed the effects of decreasing the number of individuals in the populations and the number of total iterations keeping the full RL hot restart strategy. Specifically, several tests with varying population sizes and iteration counts have been conducted for each metaheuristic to assess whether the proposed methodology could reduce the use of computational resources while maintaining the quality of the solutions.

Table 4.16 contains the mean value and standard deviation for the objective function of the best solutions found across 30 hot restart full RL runs with various parameter configurations of GA, GWO, and PSO. The cells with bold values in each approach indicate the parameters used in the first experiment. They are considered as the baseline for the comparison with the other configurations tested. The population sizes of GWO and PSO have been maintained to 20, 30, and 40 elements to have a fair comparison between them, despite the fact that in the baseline experiment the configuration of GWO included 30 wolves and the PSO configuration 40 particles. The table shows the values for GA considering population sizes of 32, 64, and 128 elements and iteration counts of 20 and 30 generations. Excluding the configuration with a population size of 128 and 30 generations, which was used in the first phase, the table shows that, with other parameter settings, GA with the full RL hot restart is able to achieve competitive solutions

Table 4.15: Hot and cold restart 20 runs outcomes on every approach under analysis.

	Cold	**60% RL 40% Meta**	**40% RL 60% Meta**	**40% RL 30% Meta 30% Random**	**Full RL**
GA	-19.86483 (1.39952)	2.43686 (0.36974)	2.72712 (0.46544)	2.72652 (0.33703)	**2.82677 (0.37890)**
GWO	0.13099 (0.86079)	**4.06706 (0.05632)**	4.06351 (0.09070)	4.06160 (0.06615)	4.05947 (0.07684)
PSO	-6.02489 (1.57020)	3.99443 (0.11492)	4.00392 (0.10494)	4.01084 (0.12876)	**4.04345 (0.10680)**

Table 4.16: Hot restart outcomes for GA, GWO, and PSO on different population/swarm sizes and iteration settings.

GA (Population Sizes: 32, 64, 128)			
20	2.54829 (0.49085)	2.65621 (0.41239)	2.60675 (0.44974)
30	2.61122 (0.39611)	2.68641 (0.43390)	**2.76804 (0.38129)**
GWO (Swarm Sizes: 20, 30, 40)			
20	3.07854 (0.57118)	3.68276 (0.21434)	3.84611 (0.15012)
30	3.97358 (0.09884)	**4.05507 (0.10265)**	4.09347 (0.05825)
PSO (Swarm Sizes: 20, 30, 40)			
20	3.75920 (0.16435)	3.79052 (0.14884)	3.81225 (0.12001)
30	3.95463 (0.13264)	3.98914 (0.12070)	**4.02452 (0.12372)**

compared to the baseline. In fact, the maximum and minimum values of the objective function are very similar across all parameter configurations. This demonstrates that a hot restart applied to GA can lead to significant savings of computational resources and time without compromising its effectiveness, which still remains the worst overall if compared with other approaches. Regarding GWO, it appears to be more sensitive to parameter variations in terms of the objective function compared to GA. This is confirmed by the margin of more than 20% between the highest and lowest values of the metric considered when moving from a configuration with 40 wolves and 30 generations to a swarm of 20 wolves and 20 total generations. Finally, values related to the hot restart PSO show a trend similar to GA. In fact, the objective function maintains good stability across all the different configurations analyzed. However, its efficiency remains consistently much higher, confirming the very positive trend demonstrated in the previous evaluation and shown in Fig. 4.9. As a conclusion, PSO and GA proved to be the most stable across all parameter settings, with the PSO significantly outperforming the latter. On the other end, GWO is the approach that obtained the highest values in different configurations but resulted in more sensible than the other metaheuristics. Moreover, these experiments proved not only the advantages of using a RL-based approach as a hot restart methodology for different well-known evolutionary algorithms but also its potential resource conservation, while maintaining acceptable performance across the majority of the configurations analyzed. One final consideration can be made regarding the potential savings in resources dedicated to the training needed for the RL model. For all the experiments, only a single model trained with 50.000 steps was used, thanks to the adaptability that PPO showed previously in Section 4.1. Taking advantage of this aspect along with the usage of DT as a potential training ground, it is possible to build flexible solutions to tackle the very high dynamicity that characterizes environ-

ments like the CC.

4.4 Chapter Summary

This chapter presented service management problems where infrastructure providers need to manage a pool of services in CC environments by maximizing metrics such as VoI and PSR considering the criticality of different services, demonstrating the significant potential that RL and CI have in dealing with this kind of orchestration.

Section 4.1 showed a comparison of the performance of CI algorithms (GA, PSO, QPSO, MPSO, GWO, and their ECT variations) and DRL algorithms (DQN and PPO). To address the proposed service management problem using DRL algorithms, a MDP has been proposed in which an agent learns how to distribute service instances throughout the CC by using a custom reward that takes into account the PSR and a penalty for infeasible allocations. The experimental results in a simulated CC scenario showed that, given an adequate number of training steps, all approaches can find good quality solutions in terms of the objective function. Furthermore, the adoption of a penalty component in the fitness function of the CI algorithms was an effective methodology to drive the convergence of the CI algorithms and to improve the overall results. Then, in an experimental analysis in a what-if scenario, where a sudden disconnection of the cloud layer has been simulated, PPO retained its effectiveness even without performing another training round, presenting consistency with the first experimentation. In contrast, DQN was not capable of achieving good results. Among the CI approaches, both versions of GA had a significant drop in the maximization of the objective function. On the other hand, all variants of PSO found competitive solutions compared with DRL algorithms. However, all of them needed a new training phase to achieve that performance, making them more costly and less adaptive in highly dynamic scenarios.

Section 4.2 introduced and validated VOICE, a cutting-edge platform designed for service management in CC ecosystem. The concepts and design of the VOICE platform showed that VoI-based service development and management has the potential to enable the creation of services that can seamlessly adapt to different contexts through the CC with high dynamics, using available resources efficiently, and providing very high QoS to the end users. Thanks to a VoI-based technique that emphasizes the significance of data filtering, VOICE supports dynamic service components allocation throughout the CC. To evaluate these principles as enabling technology, CI approaches have been compared with more traditional strategies (e.g. maximization of satisfied requests or latency minimization). Final results showed how VoI can be a valuable proposal to optimize the use of resource scarcity that characterizes CC environments.

Finally, Section 4.3 presented a hybrid hot restart technique that tries to combine the strengths of CI and RL tools to re-optimize the deployment configuration when a

reconfiguration is required by the dynamic conditions of the CC. The proposed hot restart methodology makes use of PPO to create a starting population for metaheuristics, which can further improve results by exploring the proximity of the search space. Experimental results showed how the proposed method can be used to effectively find a new system configuration when the environment changes. Furthermore, hot restart can limit the number of objective function evaluations within the CI approach, manipulating parameters such as population size or number of iterations without losing significant effectiveness.

Chapter 5

Kubernetes for the Compute Continuum Management

5.1 Kubernetes Fundamentals

K8s is a container orchestrator designed to automate the deployment, scaling, and management of containerized applications. It handles the entire lifecycle of container-based workloads: launching and monitoring applications, performing rolling updates, maintaining service availability, dynamically scaling workloads on demand, and enforcing multiple layers of security policies. Originally released as an open-source project on GitHub in 2014, K8s has since grown into one of the most widely adopted platforms for container orchestration, supported by a global community of more than 2,500 contributors. All major cloud providers offer fully managed K8s services, enabling a uniform experience whether applications are deployed in public clouds, private data centers, or even local development machines [179].

One of the key reasons for K8s widespread adoption lies in its portability and standardization. Once an application has been properly containerized and configured, it can be deployed consistently across heterogeneous environments without significant modifications. This property makes K8s particularly appealing to organizations aiming to adopt hybrid, multi-cluster, or multi-cloud strategies, where applications must seamlessly move between different DCs or cloud vendors. Once a certain level of expertise is achieved, K8s allows projects to be developed, tested, and deployed with a high degree of efficiency and reproducibility, reducing their time to market. K8s introduces developers and operators to complex concepts at the infrastructure level, such as load balancing, service discovery, networking policies, and persistent storage. These features are continuously evolving along with their related challenges, reflecting constant innovation and evolution in the cloud-native ecosystem.

A K8s deployment is typically composed by a cluster, a set of physical or virtual

machines collectively managed by the K8s control logic. These machines, referred to as nodes, provide the computational resources on which containerized workloads are executed. A K8s cluster is composed of at least one worker node, responsible for hosting the Pods that run user workloads, and a control plane, which orchestrates the entire cluster by managing worker nodes, scheduling workloads, and maintaining the desired system state. In production setups, the control plane is typically replicated across multiple machines to ensure high availability and fault tolerance, thus preventing a single point of failure from disrupting the operation of the cluster [180].

5.1.1 Kubernetes Control Plane

The control plane acts as the brain of a K8s cluster, hosting the core components required for its management and orchestration. In managed cloud environments (e.g., Amazon Elastic K8s Service or Google K8s Engine), these components are provisioned and maintained by the provider, thus abstracting away most of the operational complexity from the user. In contrast, in self-managed or on-premises deployments, the administrator is responsible for installing, configuring, and maintaining each component to ensure that the cluster can schedule a sufficient number of containers and that the required resources remain continuously available. The control plane is composed of four fundamental components, each of which plays a critical role in the functioning of the cluster:

- **kube-apiserver**: The central management entity of K8s. It exposes a RESTful Application Programming Interface (API) over HTTPS, which acts as the main communication interface between users, operators, and internal components. Clients such as kubectl interact with the cluster through the API server, submitting workloads, retrieving status information, and performing configuration changes.
- **kube-scheduler**: Responsible for workload placement. Whenever a new Pod is created and requires execution, the scheduler determines on which worker node it should run, taking into account factors such as resource availability (CPU, memory), hardware constraints, taints/tolerations, and affinity/anti-affinity rules. K8s also allows the deployment of custom schedulers, enabling advanced use cases such as domain-specific scheduling policies or MOO strategies, discussed lately in this thesis.
- **kube-controller-manager**: The main daemon that runs the core controllers. These controllers continuously monitor the cluster state through control loops and take corrective actions to reconcile the desired state (as declared by the user) with the observed state. For instance, they manage node availability, maintain

the number of replicas defined in Deployments, and update Service endpoints to ensure proper connectivity.

- **etcd**: A distributed and fault-tolerant key–value store that serves as the single source for all cluster data. It maintains information about Pods, ConfigMaps, Secrets, and the overall cluster state. Because etcd is critical to guarantee K8s consistency, its data is replicated across multiple instances to tolerate failures and prevent data loss.

To ensure high availability of the control plane, it is recommended to deploy multiple control plane nodes. These nodes participate in a consensus protocol (typically based on Raft) that allows the cluster to elect a new leader if the current one becomes unavailable [181].

5.1.2 Kubernetes Worker Node

The worker node is responsible for executing the service components through the creation and execution of the Pods assigned to it, ensuring that their containers are running as expected and remain connected to the K8s network. Similarly to the control plane, the worker node is composed of several essential components, which together constitute the data plane of the cluster:

- **kubelet**: A node-level agent that runs directly on the host machine (not in a Pod or container). It is responsible for receiving Pod specifications from the API server, managing their life cycle, and performing continuous health checks through periodic heartbeats. The kubelet ensures that the actual state of the containers matches the desired state declared by the control plane, restarting or terminating containers when needed.

- **kube-proxy**: The network component in charge of routing traffic within the cluster. It maintains network rules that enable communication between Pods and external services. It runs as a DaemonSet, ensuring that each node has its own proxy instance. K8s supports multiple proxy modes, including userspace, iptables (the default), and IP Virtual Server, each with different performance and scalability characteristics.

- **Container runtime**: The software layer used by kubelet to create and manage containers. While K8s is most commonly deployed with containerd as the default runtime, it supports any implementation compliant with the Container Runtime Interface, including Docker Engine, CRI-O, and others.

The components above represent the fundamental building blocks of the K8s architecture, but the system is highly extensible. Additional components such as CoreDNS

(for service discovery) [182] and the cloud-controller-manager (to integrate with public cloud APIs) [183] can be added to extend functionality. Although K8s is fully open-source and can be deployed using components maintained entirely by the community, it is crucial to remain aware of the operational complexity introduced by such systems (particularly in production environments) since their proper configuration and maintenance are essential to ensure scalability, security, and reliability. To this end, Fig. 5.1 provides a big picture of the K8s infrastructure along with the main components previously described.

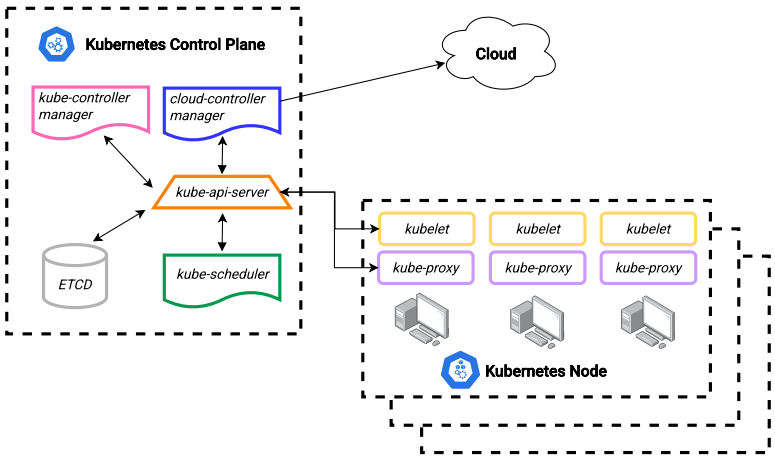


Figure 5.1: High-level figure of the Kubernetes infrastructure.

5.1.3 Kubernetes Objects

In K8s, the logical entities that define and manage workloads are called objects, which are exposed and manipulated through the Kubernetes REST API. Applications are described in a declarative way through YAML manifests, which specify the desired state of the system, such as the number of replicas, configuration settings, and resource requirements. These manifests are submitted to the API server, which stores them in the cluster's distributed key–value store (etcd). K8s continuously compares the desired state (as declared in the manifests) with the current state of the cluster, and performs the necessary reconciliation actions to align the two. This process may involve creating or terminating containers, updating configurations, or rescheduling workloads to different nodes. This declarative procedure is one of the most powerful features of Kubernetes, as it enables automation and self-healing: the system autonomously recovers from failures and drifts in state without requiring manual intervention [184].

What makes K8s a complete orchestration system is that it manages much more than just container life cycles. The distributed database of the cluster (etcd) stores not

only workload definitions but also sensitive data such as API keys and credentials, which are securely exposed to workloads through the Secret object [185]. K8s also provides persistent storage abstractions, allowing data to live outside containers and survive Pod restarts, and traffic management mechanisms to route incoming requests to the appropriate Pods, enabling scalability and high availability. Importantly, K8s is application-agnostic: it does not require knowledge of how the application is implemented internally. It can orchestrate a cloud-native microservices application distributed across dozens of Pods, or a legacy monolithic application encapsulated in a single container. This agnosticism allows K8s to serve as a unifying platform for a wide spectrum of workloads, bridging the gap between traditional software and modern cloud-native architectures.

Pods are the smallest deployable unit and represent a single instance of a running component within the cluster. Each Pod encapsulates one or more tightly coupled containers, along with their shared storage and networking context. Each Pod is assigned its own virtual IP address, which allows it to communicate seamlessly with other Pods in the cluster, regardless of the physical or virtual node on which they are hosted. The K8s virtual networking model abstracts node boundaries, enabling transparent communication throughout the cluster. In most cases, each Pod contains a single container, which simplifies deployment and resource management. However, K8s also supports multi-container Pods, where containers share the same network namespace and local resources, effectively operating within a single execution context. This configuration is particularly useful for implementing design patterns required by logging, monitoring, or proxying services for the primary application container [186].

Services are flexible resources that abstract and manage network access to a dynamic set of Pods. They provide routing capabilities for three main communication patterns: Pod-to-Pod communication within the cluster, external clients accessing Pods inside the cluster, and Pods interacting with external systems. While Pods are assigned unique IP addresses, these addresses are ephemeral: they change whenever a Pod is terminated and replaced. Directly addressing Pods by their IPs would require querying the K8s API continuously, which could be inefficient and error prone. Instead, Services provide a stable network identity, exposing a static IP and Domain Name System (DNS) name that act as an abstraction layer. Services also perform service discovery, load balancing, and health monitoring to ensure that traffic is routed only to healthy Pods. The mapping between a Service and the set of Pods is typically defined through label selectors, which dynamically associate Pods to the Service based on their metadata. This design allows seamless scaling: as new Pods matching the selector are created, they are automatically included in the Service pool, and failed Pods are excluded [187]. K8s supports multiple Service types, each suited for specific networking requirements:

- **ClusterIP**: exposes the Service only within the cluster, enabling communication exclusively between Pods. This is ideal for internal components of distributed

systems that do not require external access.

- **NodePort**: opens a static port on each node of the cluster, forwarding traffic from that port to the corresponding Service. A ClusterIP Service is automatically created as part of this setup.

- **LoadBalancer**: Integrates with external load balancers (typically provided by cloud platforms) to distribute traffic into the cluster. Requests are forwarded to a Service, which then routes traffic to the appropriate Pods, regardless of their hosting node. Both NodePort and ClusterIP Services are created automatically as dependencies.

- **ExternalName**: Acts as a DNS alias that maps a Service name to an external domain name. This allows Pods to use local DNS entries, which K8s transparently resolves to the external resource whenever a DNS lookup occurs.

Through these mechanisms, Services make distributed applications resilient, scalable, and network-transparent, hiding the complexity of ephemeral Pods while ensuring seamless connectivity across heterogeneous workloads.

Furthermore, K8s provides several built-in workload controllers that extend the basic Pod and ReplicaSet abstractions, enabling more advanced orchestration capabilities [188]:

- **Deployment** A higher-level controller built on top of ReplicaSets. While a ReplicaSet ensures that a specified number of Pod replicas are always running, a Deployment introduces advanced features such as update strategies (e.g., rolling updates or recreate strategies), rollback mechanisms, and versioned history. This makes Deployments the standard abstraction for managing stateless applications in production environments, as they combine scalability, resilience, and controlled evolution of applications over time.

- **DaemonSet**: Ensures that a copy of a specific Pod is running on all (or a selected subset of) nodes within the cluster. DaemonSets are typically used to run background daemons that provide node-level services, such as log collection agents, monitoring tools, or network proxies. By guaranteeing one instance per node, DaemonSets enable uniform and distributed functionality across the cluster infrastructure.

- **Job and CronJob**: These controllers manage workloads that are expected to terminate upon completion. Unlike Deployments, which keep Pods running continuously, Jobs are designed for batch processing and short-lived tasks. A Job ensures that a Pod successfully completes its execution, and its logs and execution results remain available for inspection after termination. Importantly, Jobs

leverage the entire cluster resource pool, making them suitable for computationally intensive tasks. CronJobs extend this concept by allowing the scheduled execution of Jobs at predefined intervals. They are commonly used for recurring operations, such as periodic backups, report generation, or automated email notifications.

Finally, this subsection discusses about K8s scaling aspects. Specifically, K8s abstracts networking and storage layers away from the compute layer, enabling Pods (i.e., replicated instances of the same application) to scale seamlessly while attaching to the same underlying abstractions. Beyond this baseline replication model, K8s provides additional autoscaling mechanisms that dynamically adjust resources according to predefined metrics, thresholds, and policies. This ensures that applications can adapt to fluctuating workloads while optimizing resource utilization [189]. More specifically:

- **Horizontal Pod Autoscaling (HPA)**: The most common scaling mechanism, which adjusts the number of Pod replicas within a Deployment or StatefulSet. The HPA operates based on metrics such as average CPU or memory utilization, within a range of replicas specified by the user. When utilization falls below the target threshold, replicas are reduced; when it exceeds the threshold, replicas are increased. This mechanism is particularly effective for stateless workloads with variable traffic patterns.

- **Vertical Pod Autoscaling (VPA)**: Instead of scaling the number of replicas, the VPA adjusts the resource requests and limits of individual Pods, effectively scaling them vertically. Observing historical and real-time resource usage, VPA can recommend or automatically apply adjustments to CPU and memory allocations. This approach is beneficial for workloads that cannot easily be replicated horizontally (e.g., stateful or monolithic applications), but require more efficient allocation of node resources.

- **Cluster Autoscaling**: Operates at the infrastructure level by monitoring the scheduler. When insufficient resources prevent pending Pods from being scheduled, the Cluster Autoscaler provisions new nodes to expand the cluster. Conversely, it can remove underutilized nodes when workloads decrease, although scale-down is not always immediate. While Cluster Autoscaling ensures high availability under load, it also introduces risks such as overly aggressive scale-up (leading to resource overprovisioning) and delayed scale-down (resulting in inefficiencies).

Together, these autoscaling mechanisms enable K8s to manage elastic workloads across diverse deployment models. They illustrate how K8s bridges application requirements with infrastructure provisioning, a capability that is particularly relevant in CC

environments, where resources may span heterogeneous and geographically distributed infrastructures.

5.2 Kubernetes Multi-Cluster scenarios management

In recent years, containers have revolutionized application deployment and life-cycle management [190]. Applications evolved from a single monolith to a complex composition of loosely-coupled microservices, resulting in remarkable improvements in deployment flexibility and operational efficiency[191]. However, managing these modern microservice-based applications requires extremely sophisticated orchestration solutions. The emergence of novel paradigms such as Fog Computing [192] and Edge Computing [193] and new use cases (e.g., autonomous vehicles [194], virtual reality services [195]) demanding computing resources closer to devices and end-users adds further complexity and puts even more pressure on popular cloud infrastructures (e.g., K8s and Red Hat OpenShift). The lack of efficient multi-cluster management features has hindered the deployment of these applications due to their stringent requirements (e.g., low latency, high bandwidth) [196].

The current literature (e.g., [20], [21]) mainly addresses single-cluster scenarios since works studying multi-cluster orchestration are still scarce. However, the scheduling problem becomes significantly more challenging in these scenarios. Managing multiple clusters adds a layer of complexity to the overall system. Each cluster has its configuration, resource constraints, and networking settings. Coordinating these diverse environments requires an efficient orchestration system. Multi-cluster environments often involve communication and data transfer between clusters. Ensuring low latency between clusters is challenging, especially when clusters are geographically distributed. In addition, the orchestrator has to determine where to deploy each microservice and decide whether to place all its instances in a single cluster or distribute them across multiple ones. An efficient strategy is crucial to choose when to distribute microservice instances across different clusters, with the aim of enhancing resource utilization and decreasing the application's latency. Only a few works [197]–[200] propose theoretical formulations or heuristics for multi-cluster orchestration, typically evaluated via simulations or small testbeds, making their applicability in popular platforms difficult.

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

- **Kubernetes Federation (KubeFed)** [201] (archived and no longer active since April 2023) was one of the first official K8s projects focused on multi-cluster scenarios designed to manage multiple K8s clusters as a single entity. It allowed for centralized control and configuration management across clusters. KubeFed

provided features for multi-cluster application deployment, scaling, and monitoring, making it suitable for scenarios requiring centralized management and control.

- **Cluster API** [202] is a K8s subproject initiated by the K8s Special Interest Group Cluster Lifecycle dedicated to streamlining the provisioning, upgrading, and management of multiple K8s clusters through declarative APIs and tooling. Leveraging K8s-style APIs and patterns, the Cluster API project automates the management of the cluster lifecycle for platform operators. It supports numerous infrastructure components such as VMs, networks, and load balancers in a consistent manner, ensuring reproducible cluster deployments across diverse infrastructure environments.

- **Open Cluster Management** [203] is a community-driven project focused on multi-cluster and multi-cloud scenarios for K8s apps. Open APIs are evolving within this project for cluster registration, workload distribution, and dynamic placement of policies and workloads.

- **Karmada** [62] is a K8s management system that enables to run cloud-native applications across multiple K8s clusters and clouds, without the need for modifications in the applications. By supporting K8s-native APIs and providing advanced scheduling capabilities, Karmada enables an open hybrid cloud K8s environment, with key features such as centralized multi-cloud management, high availability, failure recovery, and traffic scheduling.

This section strives to tackle the orchestration challenge in a multi-cluster infrastructure by proposing an RL-based GTM for efficient application deployment in K8s, a widely adopted container orchestration platform [204]. An RL environment has been developed to provide a scalable and cost-effective solution to train RL agents for the multi-cluster orchestration problem. Numerous works [205], [206] reported that online training in RL is significantly expensive for complex tasks in the network management domain. It allows training an RL agent with a valuable dataset collected over a specific time period (e.g., several days) or by creating a realistic simulation-based environment. In addition, this section leverages the capabilities of the open-source project Karmada [62], which acts as a control-plane solution for managing multi-cluster applications across hybrid cloud settings, aiming to make its behavior more adaptive and intelligent than its current one by developing new components and novel orchestration policies to accomplish more efficient multi-cluster scheduling.

The multi-cluster scenario is defined as an aggregation of multiple heterogeneous K8s clusters managed singularly by a control plane entity (Figure 5.2). It enables a dynamic methodology to develop, deploy, and manage all the distributed between the computing layers in the CC. The proposed architecture employs K8s at every layer of

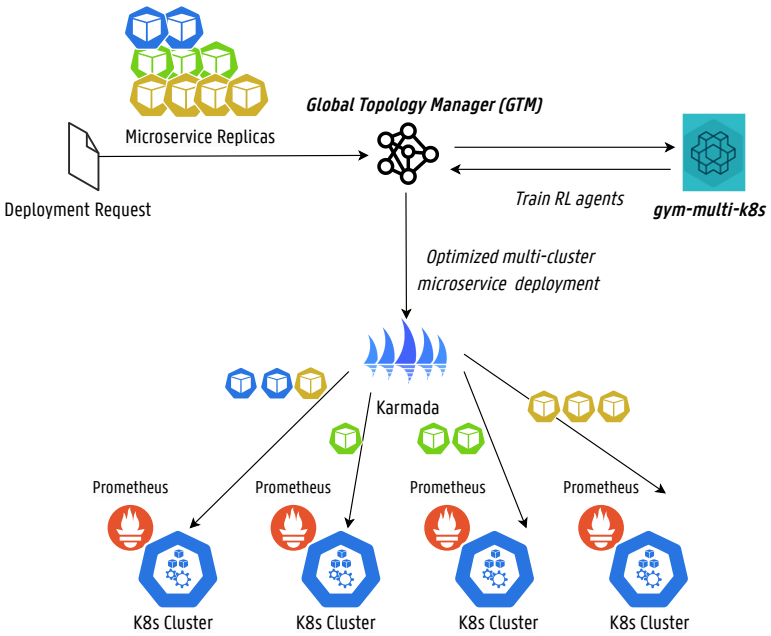


Figure 5.2: Envisioned GTM for Multi-Cluster K8s Orchestration.

the CC due to its various heterogeneous distributions such as MicroK8s, Kubedge, and K3s [61]. With a large variety of managed K8s setups, it is logical to consider this scenario as a federation of multi-cluster environments. Thus, this approach adopts Karmada [62] as a Federation Layer, a Cloud Native Computing Foundation (CNCF) project developed in continuation of KubeFed consisting of a control plane management system capable of deploying cloud-native applications across multiple K8s clusters. Its main objective is to provide autonomous management for multi-cluster applications in multi-cloud and hybrid cloud scenarios, with key features such as centralized management, high availability, failure recovery, and traffic scheduling [62]. The main reason to choose Karmada as a federation layer is that it seems a more mature solution than others available as open-source, such as Open Cluster Management. For instance, Karmada can already exploit the K8s Native API in the resource templates, making it easier to integrate with the plethora of existing K8s tools and extending it with plugins.

Despite these aspects and considerations, there is still plenty of room for improvement in the standard behavior of Karmada, especially regarding application scheduling. Karmada supports two modes to deploy replicas in a K8s cluster: *duplicated* and *divided*. The first mode implies deploying the number of requested instances in all clusters, and the second strategy splits the number of requested replicas across all clusters. Depending on the strategy favored by the cloud administrator, extra options (e.g., *ClusterAffinities*, *LabelSelectors*) can be inserted into the *PropagationPolicy* object to fine-tune the behavior of the Karmada scheduler. Karmada decides to divide replicas

mainly by the resource availability of each cluster, but typically does not consider fragmented resources leading to suboptimal scheduling. Also, the duplication policy typically leads to resource wastage, since the demand is lower than the number of reserved resources. The proposed GTM aims to automate microservice deployment in multi-cluster scenarios by finding an optimal balance between deploying the number of requested replicas in a single cluster or distributing them across several ones. Two opposing orchestration policies have been developed focused on resource efficiency and network latency to find near-optimal multi-cluster placement for different application scenarios.

The GTM communicates directly with Karmada, influencing the operations performed by the Karmada Controller Manager. Consequently, the Karmada Controller manager selects the correct controller that manages the corresponding resources of the underlying clusters through their API servers. For example, the Policy Controller monitors the deployed *PropagationPolicy* objects by creating *ResourceBinding* objects for each resource object of the group that matches the *ResourceSelector* field. The necessary deployment information for the GTM is given by extended *PropagationPolicy* objects, describing the requirements of the deployed services to allow for comprehensive evaluation of their performance requirements and adapt the resource allocation dynamically. In addition, Prometheus is applied as a monitoring agent since it provides a higher level of visibility into workloads, APIs, and distributed applications running in the cluster. The GTM takes advantage of monitoring information from Prometheus on cluster resource availability at every moment to make efficient scheduling decisions.

5.2.1 The *gym-multi-k8s* Framework

An OpenAI Gym-based framework [207] named *gym-multi-k8s* has been developed to train the RL-based GTM in a scalable and cost-efficient manner[1]. The framework enables RL agents to learn how efficiently deploy microservices in multi-cluster scenarios. The environment consists of a discrete-event RL scenario to reenact the behavior of multiple deployment requests for a given microservice deployed via Karmada on several K8s clusters. The deployment requirements (e.g., CPU and memory requests) and the number of available resources in each cluster are updated during training based on the scheduling actions of the agent. In addition, the proposed approach adopts the DS methodology presented in 3.2.3. Deep RL methods based on Multi-Layer Perceptrons (MLPs) operate in fixed-length vector spaces, which cannot support variable input and/or output dimensionalities. In other words, for the microservice scheduling problem, if an MLP-based RL agent learns on a multi-cluster setup with four clusters, it cannot be directly applied to another multi-cluster scenario that manages eight clusters.

[1] J. Santos, M. Zaccarini, F. Poltronieri, et al., "Efficient microservice deployment in kubernetes multi-clusters through reinforcement learning," in NOMS 2024-2024 IEEE Network Operations and Management Symposium, 2024, pp. 1–9. doi: 10.1109/NOMS59830.2024.10575912.

Table 5.1: The structure of the Observation Space.

Set	Metric	Description
App	R	The number of requested replicas.
	ω_{cpu}	The CPU request of each replica.
	ω_{mem}	The memory request of the replica.
	Δ	The latency threshold of the request.
	T	The expected execution time of the request.
Cluster	Π_{cpu}	The cluster's cpu capacity.
	Π_{mem}	The cluster's memory capacity.
	Θ_{cpu}	The CPU allocated in the cluster.
	Θ_{mem}	The memory allocated in the cluster.
	δ_c	The average latency of cluster c.

Table 5.2: The hardware configuration of each cluster based on Amazon EC2 On-Demand Pricing [208].

Cluster Type	Amazon Cost ($/h)	Cost (τ_c)	CPU	RAM
Cloud	t4g.2xlarge (0.2688)	16.0	8.0	32.0
Fog Tier 2	t4g.xlarge (0.1344)	8.0	4.0	16.0
Fog Tier 1	t4g.large (0.0672)	4.0	2.0	8.0
Edge Tier 2	t4g.medium (0.0336)	2.0	2.0	4.0
Edge Tier 1	t4g.small (0.0168)	1.0	2.0	2.0

Instead, DS assume that inputs and outputs can be arbitrarily-sized sets, meaning that the learned policy by the RL agent is not bound to a fixed number of clusters. Because of this, a DS-based RL agent can generalize its learned policy to different multi-cluster scenarios without retraining. The aim is that the proposed GTM, by applying DS, generalizes well to problem sizes larger than training, which would be beneficial for cloud providers scaling their infrastructure by adding additional computing power.

Table 5.1 shows the observation space considered for the multi-cluster orchestration problem, describing the environment at a given step. It includes two sets of metrics: *App* and *Cluster*. The first set *App* corresponds to the deployment requirements of the microservice-based application, such as the number of requested replicas (R), and its CPU and memory requests (ω_{cpu} and ω_{mem}). Each request also has a latency thresh-

Table 5.3: The structure of the Action Space.

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

old, which the cluster hosting the request should respect. The second set *Cluster* corresponds to the current status of the infrastructure in terms of resource capacity (Π_{cpu} and Π_{mem}), the current amount of allocated resources (Θ_{cpu} and Θ_{mem}), among others. Also, the cluster latency consists of several latency metrics depending on the number of available clusters in the multi-cluster setup, translating into C latency metrics for each cluster. Latency values are represented as values in [1.0, 500.0] milliseconds. Table 5.2 shows the resource capacities for each cluster based on different cluster types and their corresponding deployment cost. Resource capacities are then represented as values in [2.0, 32.0], and allocated resources are initiated as values in [0.0, 0.2] since each cluster has a reserved amount of resources for background services (e.g., monitoring). This information helps the agent select the appropriate actions at a given moment.

Table 5.3 shows the action space designed for *gym-multi-k8s* as a discrete set of possible actions, where a single action is chosen at each timestep. Given a deployment request, the GTM can decide to allocate the total number of requested replicas to a single cluster, divide the number of instances across all available clusters, or reject the request. Nevertheless, the size of the action space depends on the total number of clusters in the multi-cluster scenario. Let's assume the multi-cluster setup consists of C clusters, the action space length is then $C + 2$. Rejection is allowed since computational resources might be scarce at a certain moment, and no cluster can satisfy the request. The agent should not be penalized in these cases. Regarding penalties (i.e., negative reward), a simple approach commonly followed in the literature [209] is to penalize the agent if it selects an invalid action since these are typically known beforehand based on the allocated computing resources. In contrast, action masking [210] can teach the agent that depending on the current state s specific actions are invalid. This approach has recently shown significantly higher performance and sample efficiency than penalties. The action masks for each cluster c in state s can be defined as follows:

$$mask(s)[c] = \begin{cases} true, & \text{If cluster } c \text{ has enough resources.} \\ false, & \text{Otherwise.} \end{cases} \tag{5.1}$$

Whereas for spread and reject actions, the action mask is always $true$, avoiding the lock in case all actions were marked invalid. It is noteworthy that the current Karmada

does not make this decision between *deploy-all* and *spread* placement. The cloud administrator decides by indicating the preferred strategy in the *PropagationPolicy* object. The GTM aims to find the optimal balance between both policies by following a First Fit Decreasing (FFD) approach for the *spread* action (Algorithm 4). This balance depends on the selected reward function, in which two opposing strategies were designed for the GTM. The FFD heuristic sorts the clusters in descending order based on their capacity in terms of computing resources. For each cluster, a minimum factor (f) is calculated based on the minimum ratio of available CPU and memory available to the CPU and memory requested by the replica. This determines how many replicas can fit in each cluster based on the most limiting resource (CPU or memory). A threshold (Δ) is updated to the minimum value of f across all clusters, ensuring Δ reflects the ability of the most constrained cluster to accommodate replicas. Then, the algorithm iterates over the sorted clusters, placing replicas while respecting both CPU and memory constraints. If replicas are still to be deployed after the first distribution loop ($DistLoop$), the algorithm repeats the loop until all replicas are placed. Regarding the sorting criteria, several FFD heuristics have been tested by the authors, including those based on free CPU, free memory, and a combination of both with equal weighting. After analyzing the results, prioritizing free CPU yielded higher average rewards for the RL agent. Thus, free CPU has been selected as the sorting function in the FFD implementation. FFD is widely used for its simplicity and effectiveness, often yielding near-optimal solutions with a time complexity of $O(c \log c)$, where c is the number of clusters.

Two reward functions have been designed based on different objectives: cost-aware (5.2), and latency-aware (5.3). The cost-aware function leads the agent to deploy requests in clusters focused on minimizing the allocation cost (i.e., τ_c). The cloud type is significantly more expensive than fog and edge types, so the agent will prefer to deploy requests to the edge or fog since it receives a higher reward. The deployment cost of request is given by c while the maximum allocation cost is given by max. The agent is penalized if the request is rejected, and computing resources are available even when the agent supports action masking. The latency-aware function aims to satisfy the latency threshold (i.e., Δ) of each request. If the agent chooses a cluster that meets the latency requirements, it receives a positive reward (i.e., +1). In contrast, the agent is penalized if the threshold is not respected. The request's latency (l) corresponds to the average latency of the cluster considering all the latency metrics.

$$r = \begin{cases} max - c & \text{if req. is accepted} \vee (\text{req. is rejected} \wedge \\ & \text{no resources.}) \\ -1 & \text{if req. is rejected} \wedge \text{avail. resources.} \end{cases} \tag{5.2}$$

$$r = \begin{cases} 1.0 & \text{if } l \leq \Delta \vee (\text{req. is rejected} \wedge \text{no resources}). \\ -1 & \text{if } l > \Delta \vee (\text{req. is rejected} \wedge \text{avail. resources}). \end{cases} \tag{5.3}$$

Algorithm 4 FFD for spread placement

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

if $R = 1$ **then**
  $penalty \leftarrow$ true ▷ Penalize the agent
  return $\alpha = 0$
end if
$min \leftarrow 1, max \leftarrow R, \Delta \leftarrow R$ ▷ Get min and max replicas
for each $c \in C$ **do** ▷ Calculate min factor
  $f \leftarrow \min(\Omega_{\text{cpu}}[c]/\omega_{\text{cpu}}[c], \Omega_{\text{mem}}[c]/\omega_{\text{mem}}[c])$
  $\Delta \leftarrow \min(f, \Delta)$
end for
if $\Delta \geq R$ **then**
  $\Delta \leftarrow R - 1$ ▷ To really distribute replicas
end if
$S \leftarrow \text{sorted}(\Omega_{\text{cpu}})$ ▷ Sort by decreasing order of CPU
for each $c \in S$ **do** ▷ $DistLoop$: distribute replicas
  if $R = 0$ **then**
    break
  else if $R > 0 \;\&\; \Delta < R \;\&\; (\omega_{\text{cpu}} \times \Delta < \Omega_{\text{cpu}}[c]) \;\&\; (\omega_{\text{mem}} \times \Delta < \Omega_{\text{mem}}[c])$ **then**
    $\alpha[c] = \alpha[c] + \Delta$
    $R = R - \Delta$
  else if $R > 0 \;\&\; (\omega_{\text{cpu}} < \Omega_{\text{cpu}}[c]) \;\&\; (\omega_{\text{mem}} < \Omega_{\text{mem}}[c])$ **then**
    $\alpha[c] = \alpha[c] + min$
    $R = R - min$
  end if
end for
if $R = 0$ **then**
  return α
else if $R \neq 0$ **then**
  repeat
    $DistLoop$ ▷ Repeat the $DistLoop$
  until $R = 0$
end if

Multiple agents have been evaluated in the *gym-multi-k8s* environment. Most of these algorithms have been implemented based on the stable baselines3 library [161], a set of reliable implementations of RL algorithms written in Python. The evaluation consists mainly of four agents that support discrete action spaces: Advantage Actor Critic (A2C) [211], maskable PPO [212], DS PPO [143], and DS DQN. A2C is a synchronous, deterministic algorithm that combines policy and value-based algorithms. Policy-based agents learn a policy mapping input states to output actions (i.e., actors), and value-based algorithms select actions based on the predicted value of the input state (i.e., critic). The evaluated version of A2C does not support action masking. PPO is a policy gradient method for RL vastly used today for different scenarios (e.g., robot control and video games), and maskable PPO adds support for action masking. DQN combines the classical Q-Learning RL algorithm with deep neural networks. Both DQN and PPO have been adapted to use the DS neural network architecture by modifying their standard implementations in popular RL libraries. The policy network in PPO and the Q-network in DQN have been replaced with a DS to assess its generalization capabilities.

5.2.2 *gym-multi-k8s* Evaluation Setup and Results

The C2E package provides a scalable, cloud-based IoT platform, connecting sensor style devices and processing their respective data with a DT platform. As shown in Figure 5.3, its architecture includes two main applications: Eclipse Hono and Eclipse Ditto. The first relates to an open-source framework that enables the connection of several IoT devices through remote service interfaces and the communication between them thanks to various protocol implementations (e.g., HTTP REST, MQTT). Thus, lower-end devices are connected to the back-end in the cloud to publish or report data like telemetry. At the same time, Eclipse Hono facilitates their usage to update the DT provided by Eclipse Ditto and other functional operations, such as sending commands or communicating events. It is important to note that Hono has become the subject of different studies in the last few years (e.g., [213], [214]) due to its ability to support functionalities such as device authentication and machine-to-machine management. On the other hand, Ditto presents numerous services to realize DT of IoT devices. Therefore, it enables the definition of IoT solutions without the need to manage a custom back-end, allowing users to focus only on business requirements and the implementation of their applications. Throughout the years, Ditto has been applied in several works in the literature, especially in combination with other systems and tools to realize more sophisticated IoT environments [215], [216]. The C2E could be a convenient application to test the proposed multi-cluster orchestration approach since it consists of several microservices, aiming to ease the proper life-cycle management of IoT applications. Table 5.4 shows the deployment requirements for several microservices of the C2E application applied in the *gym-multi-k8s* environment.

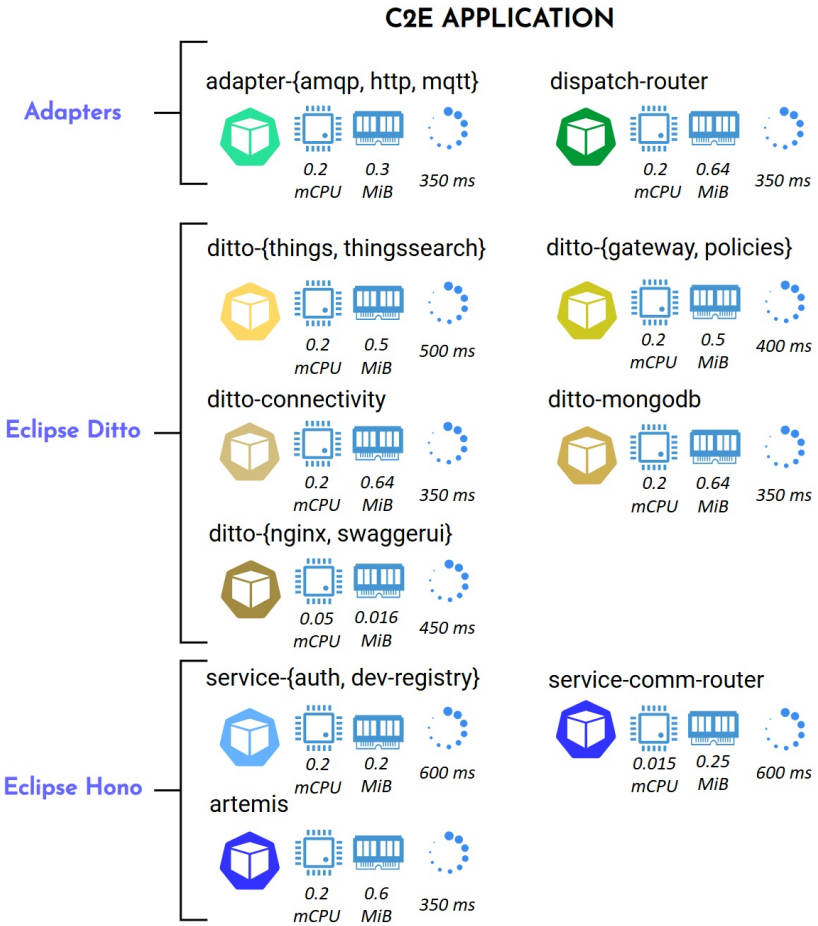


Figure 5.3: The Eclipse Cloud2Edge (C2E) architecture: processing sensor data with a DT cloud platform.

The *gym-multi-k8s* framework has been implemented in Python to ease the interaction with both the OpenAI Gym and the stable baselines 3 libraries. In the evaluation, an episode consists of 100 steps in which the agent attempts to maximize the reward based on the current deployment request. If the agent deploys the request to one of the clusters, its average latency increases as its corresponding latency metrics. In contrast, if a microservice is terminated based on a mean service duration (as default one time unit), the average latency decreases by decreasing the corresponding metrics. In addition, the action selected by the agent directly impacts the latency calculation, since a spread policy means that all clusters used for deployment will increase its latency. During training for all algorithms, the multi-cluster setup consists of four clusters. The agents have been executed on a 14-core Intel i7-12700H CPU @ 4.7 GHz processor with 16 GB of mem-

Table 5.4: Deployment properties of the C2E application.

App	Deployment	CPU	MEM	Latency Th.
C2E	adapter-{amqp, http, mqtt}	0.20	0.30	200 ms
	artemis	0.20	0.60	200 ms
	dispatch-router	0.20	0.64	200 ms
	ditto-connectivity	0.20	0.75	100 ms
	ditto-{gateway, policies}	0.20	0.50	100 ms
	ditto-{nginx, swaggerui}	0.05	0.016	100 ms
	ditto-{things, thingssearch}	0.20	0.50	200 ms
	ditto-mongodb	0.015	0.25	200 ms
	service-{auth, dev-registry}	0.20	0.20	300 ms
	service-comm-router	0.015	0.25	300 ms

ory. The performance of the agents has been evaluated based on the following metrics:

- **Accumulated reward** during each episode. It refers to the total sum of rewards obtained by an agent over time as it interacts with the environment.
- **Percentage of rejected requests** represented as $[0, 1]$. 1 means 100% rejection rate.
- **Average deployment cost** of deploying all requests in the multi-cluster setup.
- **Average latency** expected for each accepted request.

Two heuristic-based baselines have also been evaluated to compare against RL-based methods:

- **Resource-Greedy**: assigns all replicas to the cluster with the lowest resource consumption (CPU and memory).
- **Latency-Greedy**: assigns all replicas to a cluster while adhering to the specified latency threshold.

Time complexity has been accessed based on the training execution time for the multiple RL agents (Table 5.5). The results highlight that training RL agents in near-real environments is significantly faster, and that RL environments can speed up the applicability of RL methods in operational environments. A2C and Maskable PPO are considerable faster than Deepsets PPO and Deepsets DQN. **Training** results for 2000 episodes are shown in Figure 5.4 and Figure 5.5 for both reward functions. The number of available clusters during training corresponds to four for all algorithms, and

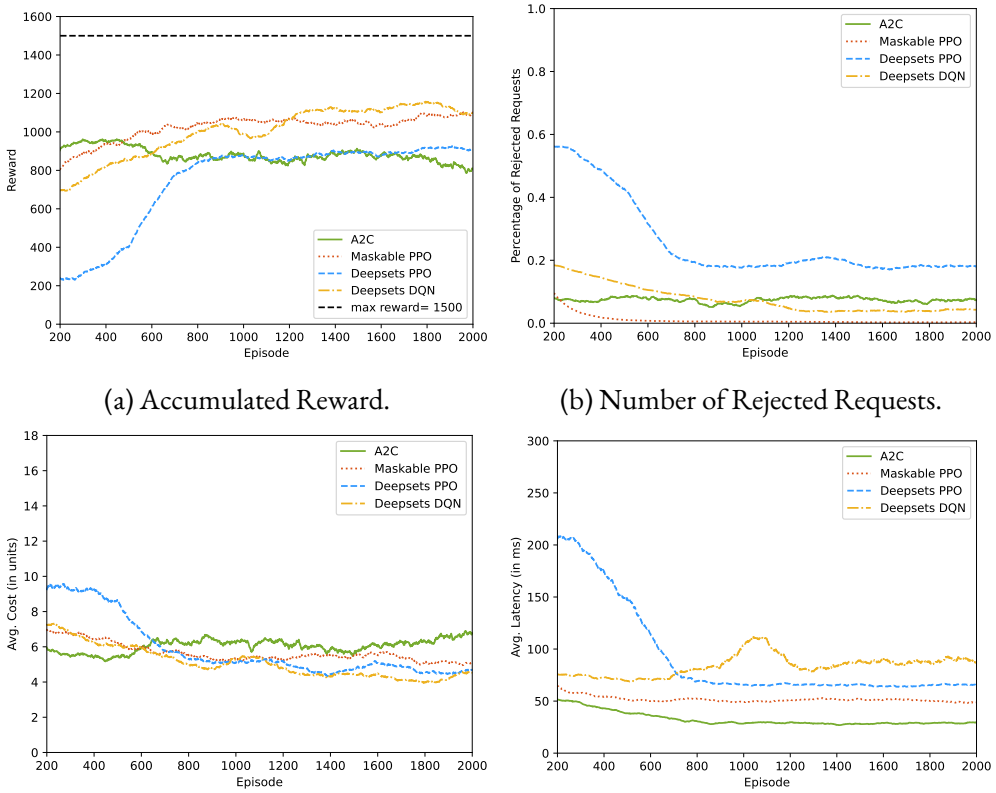
Table 5.5: The execution time per episode during training.

Algorithm	Execution Time (in s)
A2C	**0.121 ± 0.011**
Maskable PPO	0.148 ± 0.227
Deepsets PPO	0.317 ± 0.053
Deepsets DQN	0.215 ± 0.042

a smoothing window of 200 episodes is applied to reduce spikes in the graphs. Despite variations, all algorithms converge around the 1000th episode, even though DS PPO shows a dip in rewards for the latency function before surpassing previous results. All algorithms reject less than 20% of requests, with Maskable PPO reaching 0%. In terms of deployment costs, all algorithms reach average deployment costs between 6 and 12, and slightly higher values for the latency reward function, as expected. Maintaining low deployment costs while accepting a high percentage of requests is challenging with only four clusters. Lastly, regarding latency, all algorithms significantly reduce it during training, achieving average values below 50 ms, especially for the latency reward function.

Testing has been executed for all algorithms during 100 episodes with the saved configuration after 2000 training episodes. Table 5.6 summarizes the results obtained during the testing phase on the performance metrics considered for the different algorithms. Both DS algorithms achieve higher performance than A2C and maskable PPO for both reward functions, though the slightly worse training. For the cost-aware strategy, DS algorithms achieved an average accumulated reward of 1100, a low 3.9% rejection rate, and an overall deployment cost of 4.3 units. For the latency-aware function, DS-PPO achieved a 0% rejection rate with a deployment cost of 11.7 units and an average latency of 28.57 ms while DS-DQN achieved a lower deployment cost of 5.76 units on average, with a rejection rate of 0.3% and an average latency of 42.16 ms. In addition, both greedy approaches fail to provide a competitive alternative to RL methods, as they do not take into account the dynamics of the environment and cannot make tactical proactive rejections. The Resource-Greedy approach achieves lower latency on average than most cost-aware RL methods thought at a slightly higher deployment cost, while the Latency-Greedy approach achieves an average latency of 46.85 ms at a cost of a rejection rate of 2%, slightly worse than both DS algorithms.

Generalization has been assessed for both DS algorithms by varying the cluster size $[4, \ldots, 128]$. Results shown in Figure 5.6 demonstrate the enormous potential of the DS neural network. Both algorithms can find near-optimal allocation schemes for both strategies even when trained in a small-scale setup. The latency goal is considerably more complex than the cost objective while the number of clusters increases. Latency



(a) Accumulated Reward.

(b) Number of Rejected Requests.

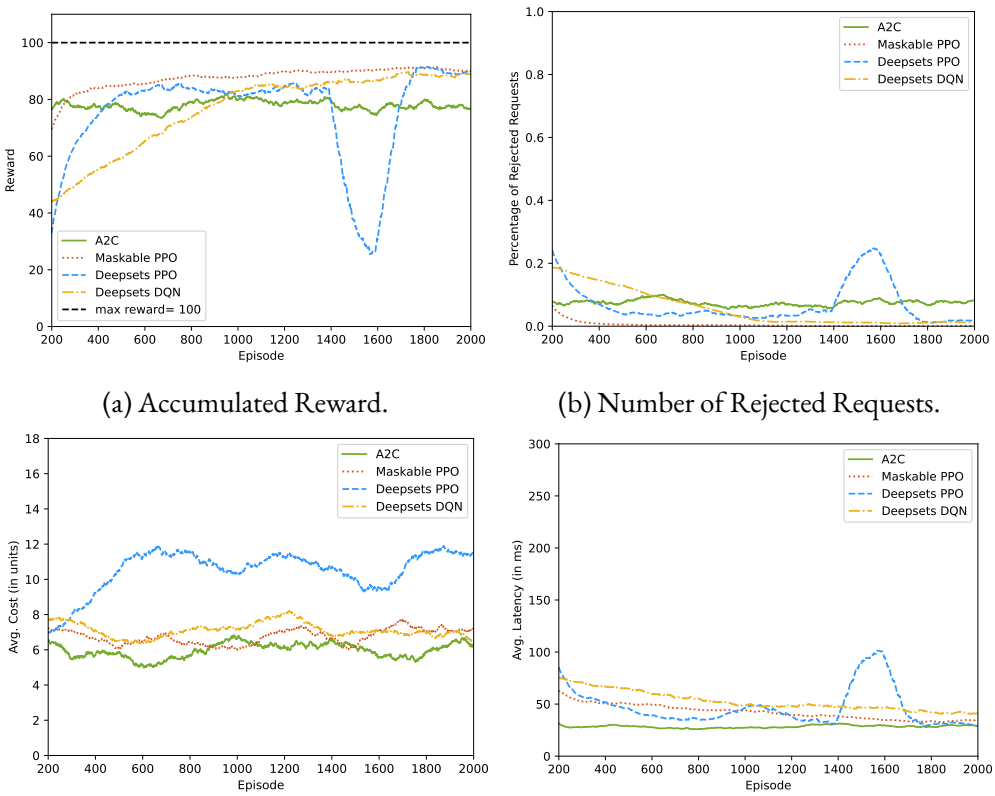
(c) Deployment Cost.

(d) Avg. Latency.

Figure 5.4: The training results for the multiple agents evaluated for the cost reward function.

increases throughout the experiment, but both agents achieve adequate latency values for both strategies, lower than 100 ms for latency-aware, and lower than 300 ms for cost-aware. DQN achieves slightly lower latency than PPO at a cost of a rejection rate of almost 4%. In conclusion, both algorithms can optimize the placement of microservices in a multi-cluster setup 32 times higher than its trained setup.

In summary, these results shown an efficient multi-cluster orchestration strategies focused on the well-known K8s platform and in recent trends as RL. Two opposing objectives demonstrate that RL algorithms can find appropriate actions that maximize the accumulated reward. An offline RL environment validated the RL approach since most algorithms achieved significantly high performance. The Karmada multi-cluster orchestration solution would benefit from this GTM since it finds a near-optimal balance between deploying all replicas into a single cluster or distributing them into multiple ones. This trade-off has been found for different objectives, as shown in this paper. In the testing phase, all algorithms achieved high rewards for both strategies. Finally,



(a) Accumulated Reward.

(b) Number of Rejected Requests.

(c) Deployment Cost.

(d) Avg. Latency.

Figure 5.5: The training results for the several evaluated agents for the latency reward function.

the DS neural network has shown its enormous potential. These RL algorithms can be applied directly to different multi-cluster environments with varying cluster sizes, significantly reducing the training time. Without DS, RL algorithms need retraining for that particular cluster size, which is considerably more costly.

5.2.3 The *HephaestusForge* Framework

Instead of considering a single-objective reward model, such as the one presented with *gym-multi-k8s* [6], *HephaestusForge* introduces a multi-objective reward function that considers three different performance factors: latency, cost, and inequality[2]. This approach enables the learning of more effective deployment strategies capable to cope with

[2]J. Santos, M. Zaccarini, F. Poltronieri, et al., "Hephaestusforge: Optimal microservice deployment across the compute continuum via reinforcement learning," Future Generation Computer Systems, vol. 166, 2025. doi: 10.1016/j.future.2024.107680.

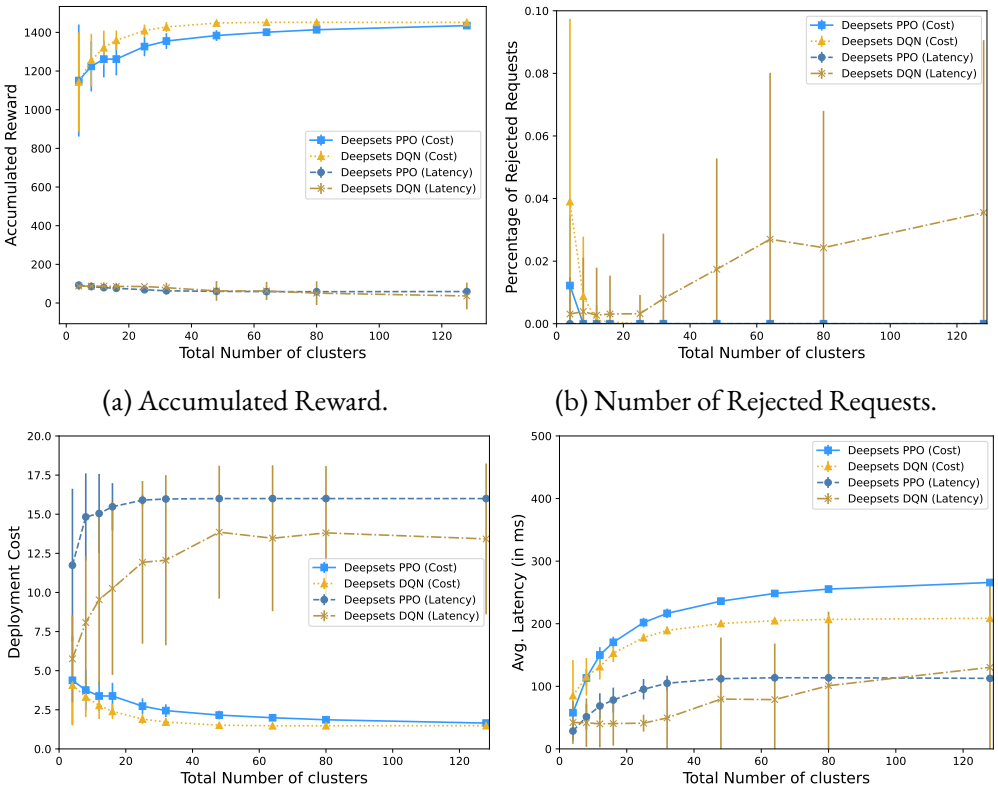
Table 5.6: Results obtained during the testing phase.

Algorithm	Reward Function	Acc. Reward	Number of Rejected Requests	Avg. Deployment Cost	Avg. latency
A2C	Cost	893.25 ± 300.23	7.30% ± 8.8%	6.30 ± 3.55 (units)	191.61 ± 53.71 (ms)
Maskable PPO	Cost	941.58 ± 183.50	9.08% ± 7.73%	5.46 ± 2.15 (units)	191.42 ± 45.85 (ms)
DS-PPO	Cost	**1150.89 ± 289.90**	**1.22% ± 2.24%**	4.38 ± 2.81 (units)	57.84 ± 17.34 (ms)
DS-DQN	Cost	1143.66 ± 256.59	3.90% ± 5.84%	**4.08 ± 2.58 (units)**	85.14 ± 56.70 (ms)
Resource-Greedy	Cost	909.22 ± 490.67	2.43% ± 4.21%	6.65 ± 5.02 (units)	**29.99 ± 15.28 (ms)**
A2C	Latency	6.04 ± 58.23	7.90% ± 13.29%	6.51 ± 5.74 (units)	183.64 ± 63.46 (ms)
Maskable PPO	Latency	33.22 ± 36.85	6.61% ± 7.89%	6.96 ± 3.55 (units)	147.79 ± 44.26 (ms)
DS-PPO	Latency	**92.64 ± 7.11**	**0.0% ± 0.0%**	11.73 ± 4.88 (units)	**28.57 ± 14.55 (ms)**
DS-DQN	Latency	87.98 ± 21.61	0.31% ± 1.18%	**5.76 ± 2.77 (units)**	42.16 ± 34.09 (ms)
Latency-Greedy	Latency	89.04 ± 9.71	2.05% ± 3.67%	6.67 ± 2.83 (units)	46.85 ± 16.29 (ms)

the several challenges that characterize CC scenarios. The results based on the implemented OpenAI Gym environment show that RL can find efficient microservice placement schemes while prioritizing latency reduction, favoring low deployment costs, and avoiding distribution inequality compared to heuristic-based methods. To do so, it considers a multi-objective reward function that addresses different performance factors and refining the RL-based approach for proper scheduling of microservice applications within the CC. *HephaestusForge* has also been open-sourced [3], enabling researchers to leverage the framework to evaluate their orchestration ideas.

In *HephaestusForge*, several RL algorithms are available for training to generate an orchestration strategy, using static and dynamic information about the K8s cluster as

[3]https://github.com/jpedro1992/HephaestusForge/tree/main



(a) Accumulated Reward.

(b) Number of Rejected Requests.

(c) Deployment Cost.

(d) Avg. Latency.

Figure 5.6: The results for the trained DS agents for both reward functions while varying the number of available clusters.

input. For example, allocated CPU and memory amounts in each cluster vary dynamically based on the number of microservice replicas deployed in each cluster. In addition, the latency of each cluster varies dynamically as if network measurements are periodically executed in the cluster, similar to recent network-aware scheduling approaches presented in the literature [217]. This information serves as input and is periodically updated after each action selected by the RL algorithm. The *HephaestusForge* framework has been developed to replicate the behavior of deployment requests via Karmada, providing the RL agent with relevant information available within a typical K8s cluster. By emulating the behavior of the Karmada scheduler, trained RL orchestration policies could then be later validated in operational environments by retrieving real-time information from the K8s cluster via popular monitoring platforms such as Prometheus [218]. However, this validation is left out of the scope of this thesis, but planned as future work. Instead, the focus is on the validation of the performance of the proposed multi-objective reward function in the *HephaestusForge* framework.

The environment consists of a discrete-event RL scenario to reenact the behavior of multiple deployment requests for a given microservice deployed via Karmada on several K8s clusters. During training, the amount of resources available in each cluster is updated based on the actions chosen by the RL agent. Similarly to the *gym-multi-k8s* framework, the observation space considered for *HephaestusForge* is the one indicated by Tables 5.2 and 5.3, as for the action space depicted in Table 5.3. Regarding the *spread* action, *HephaestusForge* follows the FFD approach defined by Algorithm 4 and sorting considerations adopted for *gym-multi-k8s*.

A multi-objective reward function (5.4) has been designed based on three different objectives: cost-aware (5.5), latency-aware (5.6), and inequality-aware (5.7). If the agent accepts the deployment request, it receives a positive reward based on these strategies and its corresponding weights (ω_l, ω_c and ω_i), normalized between $[0.0, 1.0]$. Otherwise, the agent is penalized if it decides to reject the request (i.e., -1), and computing resources were available to deploy all requested microservice replicas.

$$r = \begin{cases} \omega_c \times r_{cost} + \omega_l \times r_l + \omega_i \times r_{ineq} & \text{if req. is accepted.} \\ -1 & \text{if req. is rejected.} \end{cases} \tag{5.4}$$

$$r_{cost} = 1.0 - \Gamma_d \quad \text{where: } \Gamma_d = \text{Expected Deployment Cost for all replicas} \tag{5.5}$$

$$r_{latency} = 1.0 - \lambda_d \quad \text{where: } \lambda_d = \text{Expected Latency for deployment request} \tag{5.6}$$

$$r_{inequality} = 1.0 - G \quad \text{where: } G = \textit{Gini Coefficient} \tag{5.7}$$

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

$$\Gamma_d = \frac{1}{R} \sum_{i=1}^{R} \tau_{c_i} \tag{5.8}$$

where:

Γ_d is the deployment cost of the microservice request.
R is the number of replicas within the request.
τ_{c_i} represents the allocation cost of the i-th replica.

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

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

$$\lambda_d = \frac{1}{R} \sum_{i=1}^{R} \delta_{c_i} \tag{5.9}$$

where:

λ_d is the expected latency of the microservice request.
R is the number of replicas within the request.
δ_{c_i} represents the latency of the i-th replica.

The adoption of an average latency instead of considering multiple individual latencies lies in the need to improve the convergence speed and ease the training process of the RL agent. Specifically, the increased dimensionality of the state space and the need for the agent to track and optimize multiple individual latencies rather than a single average can significantly slow down the learning process. Although this design choice could lead to less granular performance, it favors operating with a smoother decision-making process. The study of more complex state representations will be addressed in future work. In addition, using the average latency simplifies the model and encourages

Table 5.7: The evaluated reward strategies.

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

the RL agent to minimize communication delays more equitably, avoiding excessive penalization of any single replica's latency.

Lastly, the **inequality-aware** function leads the RL agent to choose deployment actions that evenly distribute replicas across the number of available clusters. The reward is calculated based on the *Gini Coefficient* (G) [219] that ranges from $[0.0, 1.0]$, where 0 means perfect equality (all clusters host the exact number of replicas), and 1 indicates perfect inequality (all replicas deployed in one cluster). A lower *Gini Coefficient* indicates then a more equitable distribution. The *Gini Coefficient* is an accurate measure of inequality in a distribution, calculated using the formula:

$$G = \frac{\sum_{i=1}^{c} \sum_{j=1}^{c} |L_i - L_j|}{2c^2 \bar{L}} \tag{5.10}$$

where:

G is the Gini coefficient.

c is the number of clusters.

L_i is the number of replicas deployed by cluster i.

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

5.2.4 *HephaestusForge* Evaluation Setup and Results

Table 5.7 details eight distinct reward strategies considered in the evaluation of the *HephaestusForge* framework:

- ***Latency***: prioritizes minimizing latency above all other factors, assigning the highest weight ($\omega_l = 1.0$) to latency reduction. The RL agent focuses solely on reducing the expected latency of the deployment request, disregarding cost and inequality considerations.

- ***Cost***: aims to minimize deployment costs, with ω_c set to 1.0, without explicit considerations regarding latency or inequality.
- ***Inequality***: focuses on addressing inequality among clusters, with ω_i set to 1.0. The RL agent seeks to balance the deployment of replicas across clusters to mitigate disparities and ensure fair utilization.
- ***LatCost***: strikes a balance between minimizing latency and deployment costs, assigning equal weight ($\omega_l = \omega_c = 0.5$) to both objectives. The RL agent aims to achieve a compromise between latency reduction and cost efficiency in its deployment decisions.
- ***LatIneq***: balances latency reduction and inequality considerations, with $\omega_l = \omega_i = 0.5$. This strategy aims to minimize latency, while also addressing fairness across clusters.
- ***CostIneq***: prioritizes minimizing deployment costs while also addressing inequality, with $\omega_c = \omega_i = 0.5$. The goal is to achieve cost-efficient deployments while ensuring fair allocation across clusters.
- ***Balanced***: seeks a proportional approach, with moderate weights assigned to each objective ($\omega_l = 0.4$, $\omega_c = 0.3$, $\omega_i = 0.3$). The RL agent aims to achieve a compromise that considers all three factors in its deployment decisions.
- ***FavorLat***: emphasizes minimizing latency by assigning a higher weight ($\omega_l = 0.6$) compared to cost and inequality considerations ($\omega_c = \omega_i = 0.2$).

Evaluating these numerous reward strategies, the *HephaestusForge* framework aims to provide insights into the effectiveness of different allocation strategies in the deployment of microservices across CC environments. The performance of both DS-DQN and DS-PPO have been evaluated based on the same metric considered for *gym-multi-k8s* with the addition of two new ones:

- **Average CPU usage** (in %) of the selected cluster for the microservice deployment.
- **Gini Coefficient** highlighting the inequality of the deployment scheme, represented as $[0, 1]$.

Furthermore, along with the **Latency-Greedy**, three heuristic-based baselines have also been evaluated to compare against the DS-based RL methods:

- **CPU-Greedy**: assigns all replicas to the cluster with the lowest resource consumption in terms of CPU usage.

Table 5.8: The execution time per episode (ep) during training.

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

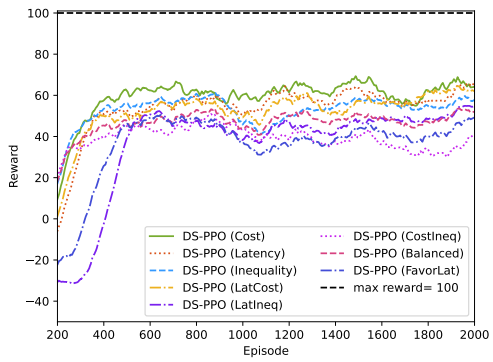
- **Binpack-Greedy**: assigns all replicas to the cluster with the highest CPU allocation.
- **Karmada-Greedy**: assigns all replicas to a cluster based on the cluster resource model (CPU and memory). It follows the default scheduling algorithm enabled in Karmada for dynamic replica assignment. Affinity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

Time Complexity has been evaluated based on the training execution time for the DS-based RL agents for the *latency* strategy as shown in Table 5.8. The results highlight that training RL agents in near-real environments can speed up the training process, and consequently the applicability of RL methods in operational environments. Both algorithms require between 16 to 27 minutes to complete training over 2000 episodes. DS-DQN is slightly faster than DS-PPO, though both algorithms are significantly slower than all heuristic baselines. On average, heuristic algorithms complete 2000 episodes in about 2 to 4 minutes since no policy training is required.

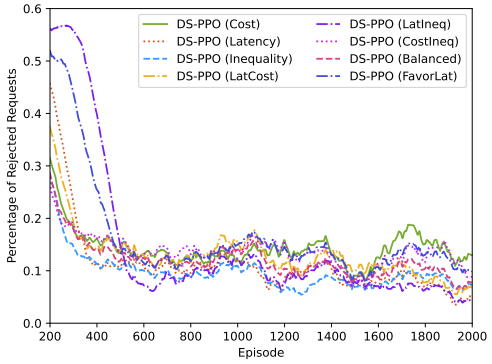
Training results for 2000 episodes are shown in Figure 5.7 and Figure 5.8 for all assessed reward strategies. A smoothing window of 200 episodes is applied to reduce spikes in the graphs. Despite fluctuations, all algorithms converge between the 700th and the 1000th episode, with some exhibiting slight improvements in rewards beyond this point. Most algorithms achieve high accumulated rewards for all reward strategies, though DS-DQN algorithms converge at a slightly higher state, resulting in higher rewards on average. It is noteworthy that all algorithms learn to minimize rejections, as reflected by their pursuit of higher rewards. PPO-based algorithms reject fewer than 20% of requests, while DQN-based algorithms reject fewer than 5%. Figures 5.7c, 5.7d, 5.8c, and 5.8d illustrate that the RL agents adhere to the favored strategy: cost-aware reward functions lead to lower deployment costs, and latency-aware reward functions result in lower latency. **Testing** has been executed for all strategies over 2000 episodes using the

Table 5.9: Results obtained by DS-RL approaches during the testing phase.

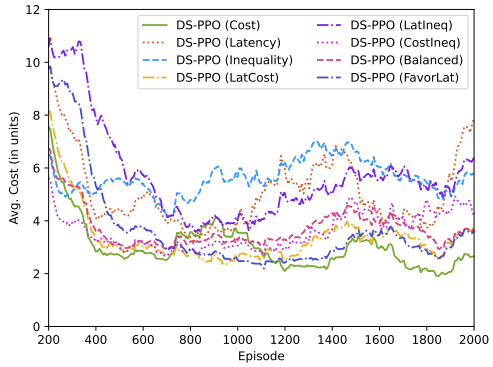
DS Alg.	Reward Function	Acc. Reward	Rejected Requests (in %)	Deployment Cost (in units)	Latency (in ms)	CPU Usage (in %)	Gini
PPO	*Latency*	73.1 ± 0.3	0.02 ± 0.01	5.87 ± 0.19	**268.6** ± 3.4	30.5 ± 0.5	0.67 ± 0.004
DQN	*Latency*	70.1 ± 0.4	0.52 ± 0.05	7.03 ± 0.19	**290.7** ± 3.8	27.5 ± 0.5	0.55 ± 0.005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



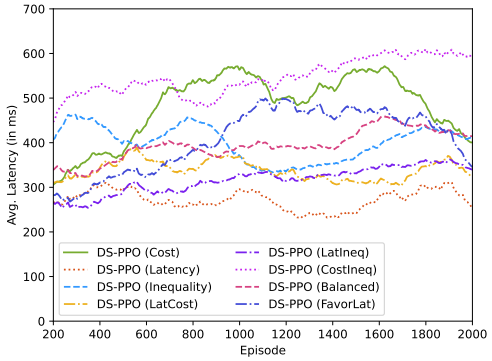
(a) Accumulated Reward.



(b) Rejected Requests.



(c) Deployment Cost.

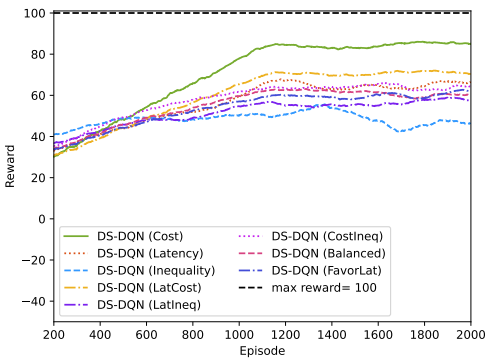


(d) Avg. Latency.

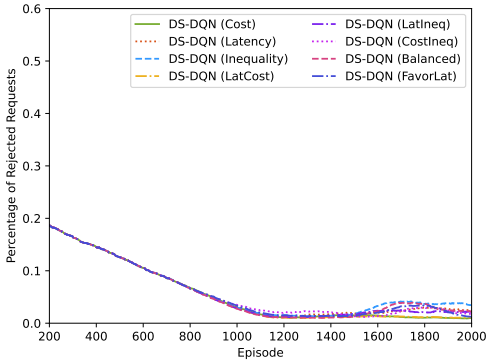
Figure 5.7: The training results for the DS-PPO agent evaluated for the multiple reward strategies.

Table 5.10: Results obtained by heuristics during the testing phase.

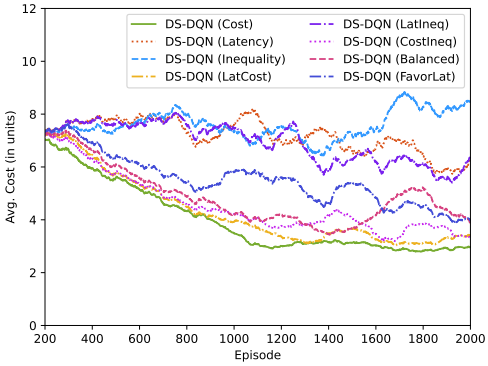
Alg.	Reward Function	Acc. Reward	Rejected Requests (in %)	Deployment Cost (in units)	Latency (in ms)	CPU Usage (in %)	Gini
CPU	-	-	**0.18 ± 0.07**	9.63 ± 0.25	422.8 ± 5.8	**18.5** ± 0.4	0.72 ± 0.003
Binpack	-	-	**0.18 ± 0.07**	**3.67 ± 0.10**	420.7 ± 5.6	34.6 ± 0.3	0.66 ± 0.004
Latency	-	-	3.74 ± 0.45	6.71 ± 0.14	**347.8 ± 2.9**	25.5 ± 0.3	**0.26** ± 0.005
Karmada	-	-	**0.18 ± 0.07**	12.25 ± 0.21	424.1 ± 5.9	**14.8** ± 0.4	0.73 ± 0.002



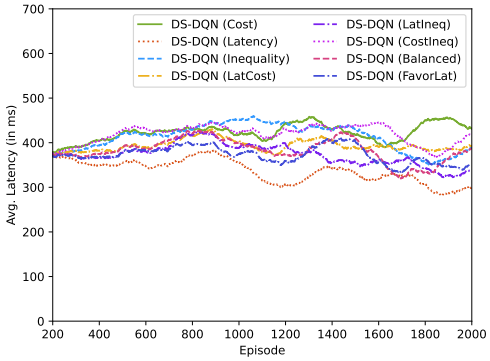
(a) Accumulated Reward.



(b) Rejected Requests.

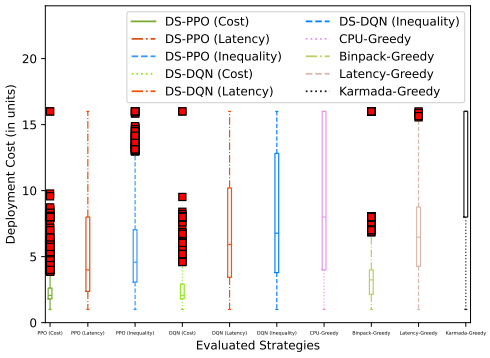


(c) Deployment Cost.

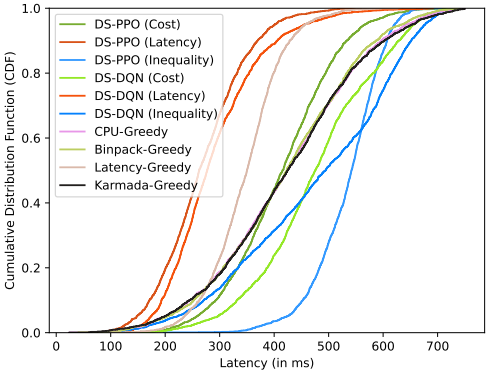


(d) Avg. Latency.

Figure 5.8: The training results for the DS-DQN agent evaluated for the multiple reward strategies.



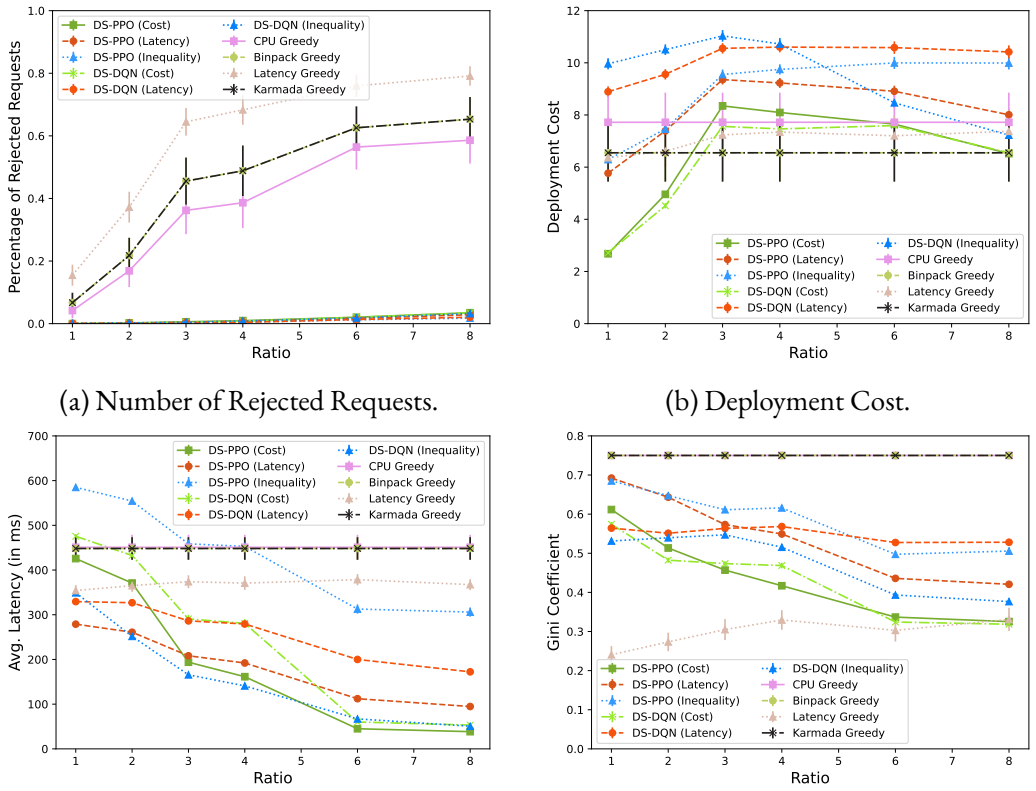
(a) Deployment Cost (in units).



(b) Avg. Latency (in ms).

Figure 5.9: The expected latency and deployment cost during testing.

saved configuration from the training phase (after 2000 episodes). Tables 5.9 and 5.10 summarize the obtained results for the different algorithms based on the considered per-



(a) Number of Rejected Requests.

(b) Deployment Cost.

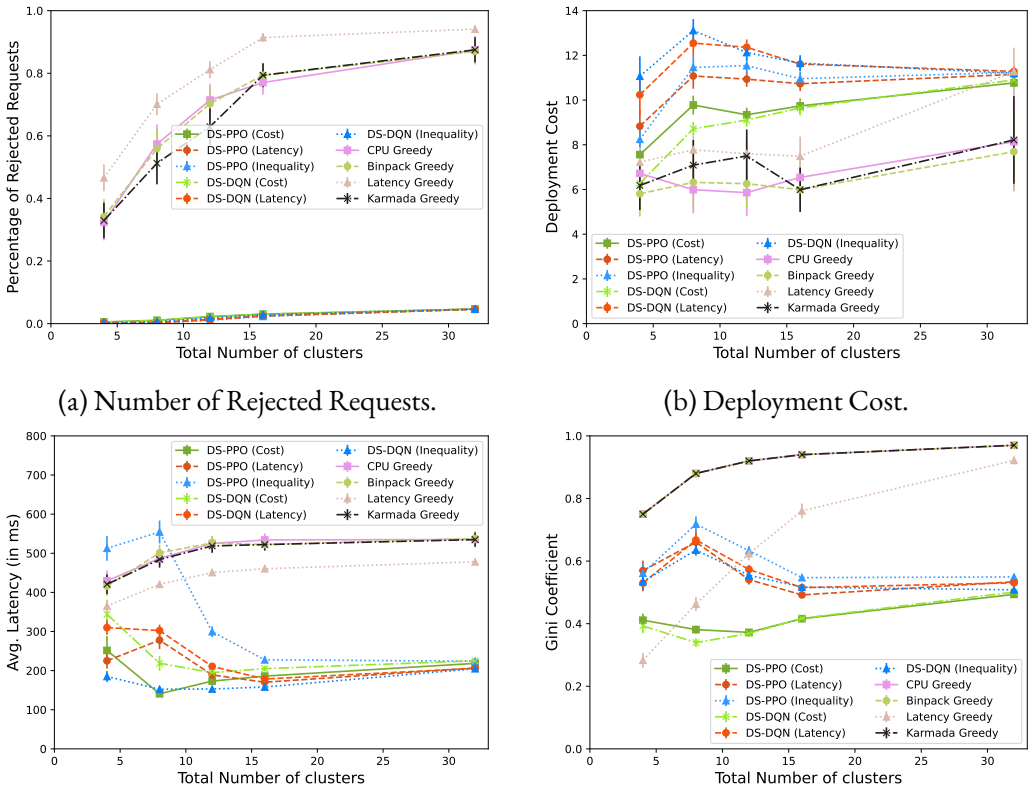
(c) Avg. Latency.

(d) Gini Coefficient

Figure 5.10: The results for the trained DS-based algorithms while varying the number of deployed replicas per request.

formance metrics. On average, latency-aware and cost-aware functions achieve higher rewards than inequality functions, highlighting the difficulty of achieving a fair distribution of microservice replicas. Most algorithms achieve a minimal rejection rate, with DS-PPO (*Inequality*) and DS-PPO (*LatIneq*) achieving as low as 0.002%. In contrast, heuristic baselines obtain higher rejection rates, ranging from 0.18% to 3.74%, because deploy-all actions might not be feasible at the end of episodes due to a lack of resources caused by previous actions. It is worth noting that DS-PPO achieves lower average rejection rates during testing compared to training. This difference occurs because, during training, the policy is stochastic: it samples actions based on the probabilities provided by the neural network. However, during testing, the policy is deterministic, selecting the action with the highest probability. This approach is standard in PPO implementations, as deterministically choosing the most probable action typically yields higher rewards, whereas stochasticity during training is necessary for effective exploration.

As expected, cost-aware reward functions result in lower average deployment costs of 2.5 units as shown in Figure 5.9a, while latency-aware reward functions lead to lower latency, ranging from 268.6 to 290.7 ms as demonstrated in Figure 5.9b. Regarding



(a) Number of Rejected Requests.

(b) Deployment Cost.

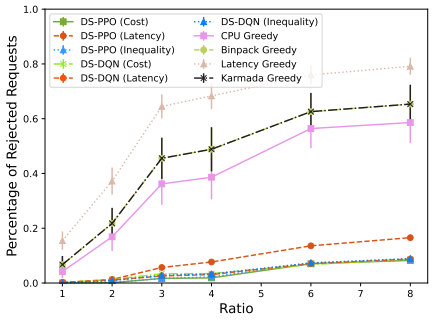
(c) Avg. Latency.

(d) Gini Coefficient

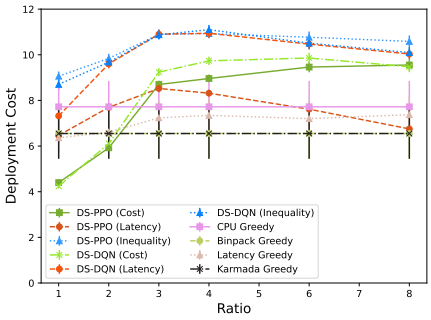
Figure 5.11: The results for the trained DS-based algorithms while varying the number of available clusters.

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

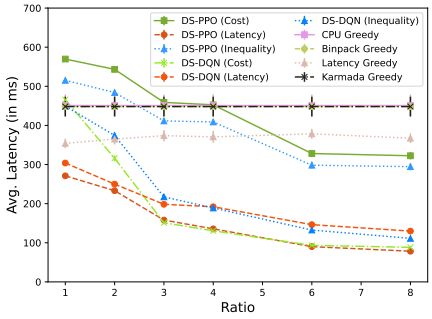
Multiple Reward Strategies have been evaluated to assess their impact on performance. Choosing an appropriate reward strategy is crucial, as it guides the learning process and significantly affects the placement strategy. As the complexity of the



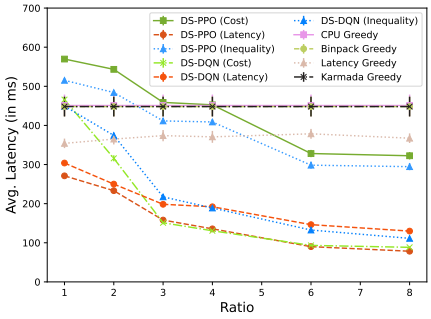
(a) Number of Rejected Requests.



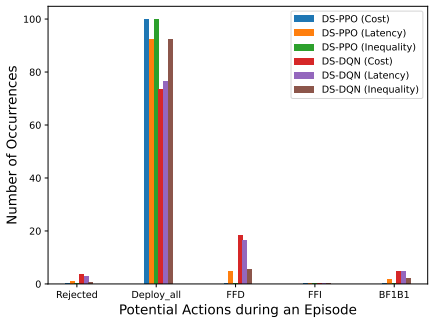
(b) Deployment Cost.



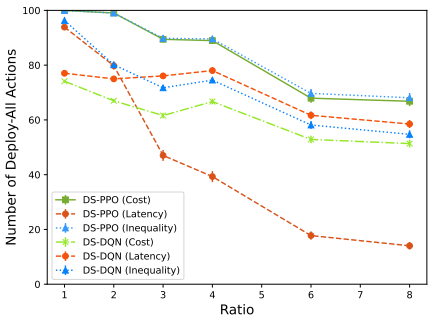
(c) Avg. Latency.



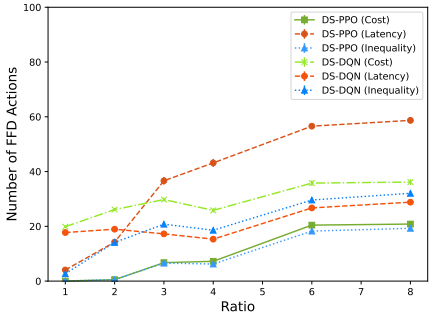
(d) Gini Coefficient



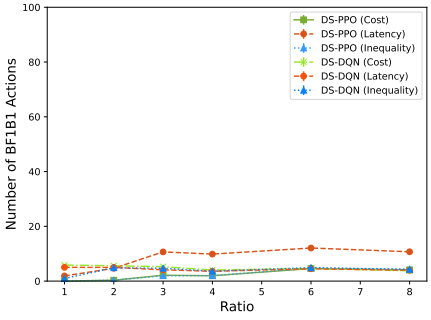
(e) Action Distribution.



(f) Deploy-all Actions.



(g) First Fit Decreasing (FFD).



(h) Best Fit (BF1B1) Actions.

Figure 5.12: The results for the trained DS-based algorithms while varying the number of deployed replicas per request for the extended version of the action space.

problem increases, it is well known that approaches such as metaheuristics can struggle, particularly when confronted with multiple performance factors, as is in this case. In contrast, this evaluation shows that RL identifies efficient policies that balance various performance factors simultaneously. However, further analysis is needed to assess how close these RL policies are to optimal solutions. Comparison of results obtained from *HephaestusForge* with those derived from exact or approximation algorithms is planned for future work, establishing benchmarks and evaluating these strategies against known optimal or near-optimal solutions.

Scalability has been assessed by increasing the number of microservice replicas per request, resulting in higher ratios during the experiments. The number of available clusters has been maintained at four, while the number of microservice replicas varies from 4 to 32, creating the following ratios [1, 2, 3, 4, 6, 8]. The minimum and maximum number of replicas has been fixed for this experiment to achieve the desired ratio. Figure 5.10 shows that all DS-based algorithms accepted most requests, even at high ratios (greater than 6). In contrast, most heuristic baselines already exhibited a 40% rejection rate at ratios of 3 and 4. In addition, as the ratio increased, most RL agents struggled to minimize deployment costs while accepting the majority of requests, although latency slightly decreased due to the higher rejection rate. The Gini coefficient also decreases with the ratio, indicating a fairer distribution as more microservice replicas need to be deployed.

Increasing the number of clusters has been evaluated by varying the cluster size from 4 to 32 and setting the minimum and maximum replicas equal to the number of clusters (c) and four times the number of clusters ($4 \times c$), respectively. This maintains a ratio between 1 and 4 throughout the experiment. The results demonstrate the enormous potential of the DS neural network. All RL agents can find near-optimal allocation schemes for all strategies, even when trained in a small-scale setup as shown in Figure5.11. As the number of clusters increases, all algorithms converge to a similar state, reflecting the challenge of optimizing cost or latency under these conditions. Nonetheless, cost-aware algorithms and latency-aware agents achieve the lowest deployment costs and lower latency, respectively. DS-based algorithms can effectively optimize microservice placement in a multi-cluster setup higher than the trained scenario, without need for retraining.

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

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

These two additional spreading strategies have been implemented and added to the action space of *HephaestusForge*, further extending the action possibilities of the RL agents. A scalability test, similar to the one previously conducted, has been carried out with an increase in the number of microservice replicas per request, resulting in higher ratios during the experiment. Figure 5.12 illustrates the performance of the RL agents and heuristic baselines during testing. Figure 5.12e shows the distribution of action selections throughout the experiment. Deploy-all actions are more commonly selected, while spreading actions are less frequent. However, based on Figures 5.12f, 5.12g, and 5.12h, spreading actions are increasingly selected as the ratio increases during the experiment. Computing resources become scarcer towards the end of episodes when the ratio is high, making the agent adapt its policy and select more spreading actions such as FFD and BF1B1. Interestingly, the agent never selects FFI, indicating a calculated avoidance of this action during testing. In terms of performance, the RL agents achieve slightly lower latency on average, slightly higher deployment costs, and a higher rejection rate compared to the RL agents tested with a smaller action space (Figure 5.10).

As a policy gradient method, DS-PPO continuously updates the policy to maximize the expected reward directly. It naturally balances exploration and exploitation by adjusting the policy based on observed rewards, allowing it to explore a wider range of actions while refining its policy more effectively. This is particularly beneficial in environments with large action space or complex dynamics, such as the multi-cluster orchestration problem where optimal actions vary based on multiple factors (e.g., latency, cost, inequality). DS-DQN, being a value-based method, estimates the value of taking a certain action in a particular state and selects actions that maximize this value. While effective, DS-DQN might struggle with exploration in high-dimensional action spaces, potentially leading to suboptimal performance if the state-action space is not fully explored. This can be a limiting factor when dealing with complex objectives such as multi-objective optimization in multi-cluster environments. DS-PPO's policy optimization approach is known for its stable learning process, which can lead to faster convergence in environments with a large number of potential actions and states. This stability is crucial in scenarios where the agent must adapt to varying conditions, such as different cluster sizes or varying ratios of microservice replicas, as tested in the scalability experiments. DS-DQN might take longer to converge or might converge to suboptimal policies, especially in environments with high variability or where the policy needs to generalize across a wide range of conditions. This could explain why DS-DQN might underperform in scenarios where rapid adaptation is required. DS-PPO is inherently well suited for environments that may involve continuous action spaces or where fine-tuning actions lead to better results. Although the environment considered primarily deals with discrete actions, the flexibility of DS-PPO in handling different types of actions might give it an edge in situations where more granular control over decisions is beneficial. DS-DQN is primarily designed for discrete action spaces, and while it can be extended to handle continuous actions, the original formulation might be less flexible

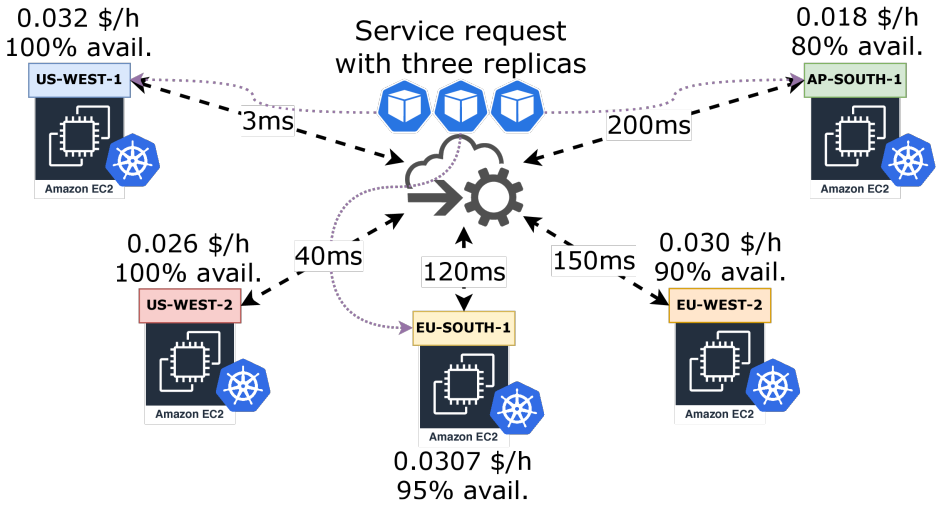


Figure 5.13: Orchestration of a service request in a multi-cluster CC scenario. Each request is associated with a geographical area corresponding to the AWS DC locations. Scheduling and resource allocation decisions are influenced by the latency and pricing of the compute instances. (Numbers are illustrative.)

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

5.3 Multi-Objective Scheduling and Resource Allocation of Kubernetes Replicas Across the Compute Continuum

As previously mentioned, traditional orchestration approaches have focused predominantly on optimizing a single objective, such as minimizing latency or deployment cost. However, this narrow focus often results in suboptimal decisions when considering other critical performance factors, leading to increased operational expenses and inefficient resource utilization. The orchestration problem is inherently multi-faceted, requiring the joint optimization of multiple, often conflicting, objectives, including latency, cost, energy consumption, and resource fairness [27]. This complexity calls

Table 5.11: Parameters of the MO-ILP

Parameter	Description
D	Set of service requests
D_t	Set of service requests that are **active** at time-slot t
DC	Set of data centers
I	Set of compute instances available in each data center
$I_r \subset I$	Set of RESERVED instances available in each DC
$I_o \subset I$	Set of ON-DEMAND instances available in each DC
$I_s \subset I$	Set of SPOT instances available in each DC
$P^r_{j,k}$	Total price for RESERVED instance $j \in I_r$ in data center $k \in$ DC instance over the complete time horizon
$P^o_{j,k}$	Hourly price for ON-DEMAND instance $j \in I_o$ in data center $k \in$ DC
$P^s_{j,k}$	Hourly price for SPOT instance $j \in I_s$ in data center $k \in$ DC
R_i	Set of replicas required by request $i \in D$
T	Set of time-slots over the scheduling period
c_i	CPU required by one replica of request $i \in D$
m_i	RAM required by one replica of request $i \in D$
C_j	vCPUs offered by instance $j \in I$
M_j	RAM offered by instance $j \in I$
$f_{j,k}$	Interruption frequency of SPOT instance $j \in I_s$ in data center $k \in$ DC
$l_{i,k}$	Latency between request $i \in D$ and data center k

for MOO techniques that can effectively steer trade-offs among these dimensions. Although linear scalarization methods are commonly employed to reduce the problem's dimensionality [221] and demonstrate significant promises (as proved with *Hephaestus-Forge* results in 5.2.4), they frequently fail to capture the diversity of trade-offs present in the solution space. Furthermore, it is complex to express a priori a quantitative preference between heterogeneous objectives, let alone simple linear combinations. In contrast, MOO offers a more expressive decision-making framework by identifying the PF, which represents the set of non-dominated solutions that achieve optimal balance across competing objectives. In this way, it is possible to present to the decision-maker a set of multiple "optimal" trade-offs, such that well-informed decisions can be made a posteriori [63], [64].
This section presents how state-of-the-art MO-EAs can be employed and shows their effectiveness in solving a MOO problem in a CC environment which includes three competing objectives: service latency, service deployment costs, and frequency of service interruption[4]. Specifically, it jointly considers the following four main aspects:

[4]N. Di Cicco, F. Poltronieri, J. Santos, et al., "Multi-objective scheduling and resource allocation of

Table 5.12: Decision variables of the MO-ILP

Variables	Description
$r_{j,k}$	1 if RESERVED instance $j \in I_r$ in data center $k \in$ DC is active, 0 otherwise.
$o^t_{j,k}$	1 if ON-DEMAND instance $j \in I_o$ is active at time-slot $t \in T$ in data center $k \in$ DC, 0 otherwise
$s^t_{j,k}$	1 if SPOT instance $j \in I_s$ is active at time-slot $t \in T$ in data center $k \in$ DC, 0 otherwise
a^t_i	1 if request $i \in D$ is scheduled to start at time-slot $t \in T$, 0 otherwise.
$x^{t'}_{i,j,k,r}$	1 if K8s replica r of request $i \in D$ is scheduled to start at time-slot $t' \in T$ and is allocated to the instance $j \in I$ in data center $k \in$ DC, 0 otherwise.
$d_{i,k}$	1 if request $i \in D$ is assigned to data center $k \in$ DC, 0 otherwise
L_i	Maximum latency experienced by request $i \in D$

- Simultaneous optimization of both scheduling and resource allocation.
- Assumption that the decision-maker's preferences between objectives are unknown before optimizing.
- Different pricing and latency models between heterogeneous clusters, profiled from real-world AWS instances.
- Possibility of splitting multiple microservices over different compute instances during resource allocation.

The scenario considered is illustrated in Fig. 5.13, where a service provider must schedule and assign compute resources to a set of service requests over the CC, considering multiple DCs and instance types. Specifically, it is assumed that requests consist of multiple replicas of microservices, which can possibly be distributed over different K8s clusters in the CC and coordinated via a global topology manager [6]. Formally, the optimization problem is modeled as a MO-ILP. The MO-ILP is given as input *i)* a set of deployment requests, characterized by CPU and RAM requirements, a request duration, and a required number of microservice replicas, and *ii)* a set of potential compute resources, representing RESERVED, ON-DEMAND or SPOT cloud instances, each characterized by a specific data center location, pricing model, hardware resource availability, and average frequency of service interruption. The objective of ILP is to decide on a scheduling and admission control rule that jointly optimizes the acceptance rate, hardware costs, and average reliability. Tables 5.11 and 5.12 illustrate the parameters and decision variables, respectively, of the MO-ILP formulation.

kubernetes replicas across the compute continuum," in 2024 20th International Conference on Network and Service Management (CNSM), 2024, pp. 1–9. doi: 10.23919/CNSM62983.2024.10814307.

Table 5.13: Hardware configuration of each cluster based on Amazon EC2 On-Demand and Spot Pricing [208], [220].

DC	Instance	On-Demand ($/h)	Reserved On-Demand ($/h)	Spot ($/h)	vCPU	RAM	Spot Interrupt Frequency
eu-south-1 (Milan)	t4g.2xlarge	0.3072	0.1842	0.0308	8	32.0	10%
	t4g.large	0.0768	0.0461	0.078	2	8.0	15%
	t4g.small	0.0192	0.0115	0.0019	2	2.0	10%
eu-west-2 (London)	t4g.2xlarge	0.3008	0.1808	0.01148	8	32.0	5%
	t4g.large	0.0752	0.0452	0.0267	2	8.0	10%
	t4g.small	0.0188	0.0112	0.0061	2	2.0	5%
us-east-1 (N. Virginia)	t4g.2xlarge	0.02688	0.1606	0.1135	8	32.0	15%
	t4g.large	0.0672	0.0402	0.0268	2	8.0	20%
	t4g.small	0.0168	0.0100	0.0090	2	2.0	5%
us-west-1 (N. California)	t4g.2xlarge	0.3200	0.1918	0.0849	8	32.0	10%
	t4g.large	0.0800	0.0480	0.0248	2	8.0	10%
	t4g.small	0.0200	0.0120	0.0061	2	2.0	5%
ap-northeast-3 (Osaka)	t4g.2xlarge	0.3482	0.2089	0.0415	8	32.0	10%
	t4g.large	0.0879	0.0522	0.0298	2	8.0	15%
	t4g.small	0.0218	0.0130	0.0031	2	2.0	15%
ap-south-1 (Mumbai)	t4g.2xlarge	0.1792	0.1080	0.0760	8	32.0	10%
	t4g.large	0.0448	0.0270	0.0211	2	8.0	15%
	t4g.small	0.0112	0.0068	0.0049	2	2.0	5%

Table 5.14: Average latency (ms) between AWS EC2 DCs [176].

DC	eu-s1	eu-w2	us-e1	us-w1	ap-n3	ap-s1
eu-s1	2.57	27.69	102.16	162.51	231.99	110.53
eu-w2	26.92	3.88	77.57	147.85	217.17	119.41
us-e1	102.04	78.57	5.15	64.21	154.10	195.66
us-w1	162.10	147.91	63.61	3.24	109.47	231.06
ap-n3	232.13	217.81	153.88	110.11	2.32	131.98
ap-s1	109.98	119.64	193.90	231.79	131.34	3.23

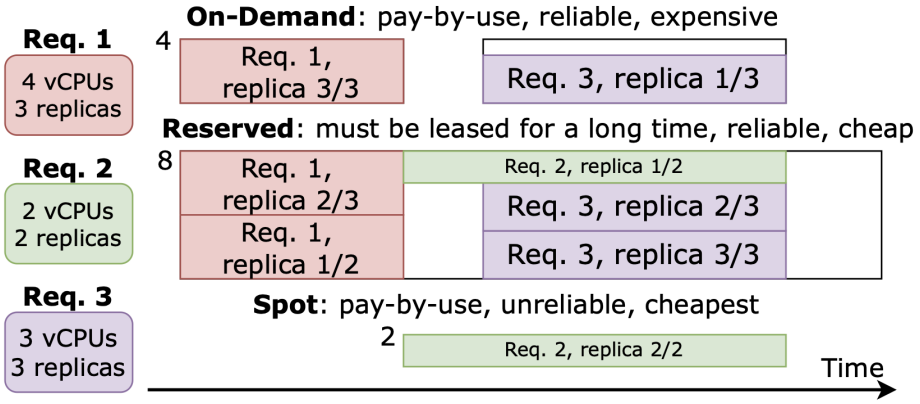


Figure 5.14: Illustrative feasible solution MO scheduling and resource allocation of K8s replicas, considering only vCPUs for ease of visualization. Replicas may be split across multiple compute instances, each with different characteristics, and must start at the same time-slot.

The first goal is to minimize the maximum delay experienced by each request, which is expressed as follows:

$$\min f_1 = \min \frac{1}{|D|} \sum_{i \in D} L_i. \tag{5.11}$$

The second goal is to minimize the total cost of leasing the hardware instances. Specifically, a realistic pricing model inspired by AWS EC2 [208], [220] is adopted, considering three types of cloud computing instances, namely Reserved, On-Demand, and Spot. Reserved instances offer a low hourly rate, but require reserving capacity for one or multiple years. For this reason, in the case a Reserved instance is activated, its cost is fixed respectively from usage, and amounts to the total cost sustained over the considered scheduling time period. On-Demand instances offer a higher hourly rate than Reserved instances, but can be instantiated on the fly (e.g., to sustain a small amount of demands, which would not justify instantiating a Reserved instance) and

are paid only for the amount of time they are used. Finally, SPOT instances are the cheapest available instances. These offer unused capacity on the cloud at a highly discounted rate compared to RESERVED and ON-DEMAND instances, can be instantiated on the fly, and are paid by their usage. However, they offer a low level of reliability, since the allocated capacity is not guaranteed but can be requested back by the cloud provider. By considering these instance types and their pricing models, the cost minimization is expressed as follows:

$$\begin{aligned}\min f_2 = \min & \sum_{k \in \mathrm{DC}} \sum_{j \in I_r} P^r_{j,k} r_{j,k} \\ & + \sum_{k \in \mathrm{DC}} \sum_{t \in T} \sum_{j \in I_s} P^o_{j,k} o^t_{j,k} + \sum_{k \in \mathrm{DC}} \sum_{t \in T} \sum_{j \in I_s} P^s_{j,k} s^t_{j,k}\end{aligned} \tag{5.12}$$

Note that cost minimization competes with latency minimization. In general, the cheapest cluster instances are not necessarily the ones offering the lowest latency to a request. For example, with reference to the parameters illustrated in Tables 5.13 and 5.14, consider a set of requests coming from the EU-SOUTH-1 area. Suppose that all requests are assigned to ON-DEMAND instances. Deploying them in AP-SOUTH-1 instead of the closer EU-SOUTH-1 instances would decrease the hourly rate for a single ON-DEMAND instance by 1.7x, but would increase the average latency by 43x. In contrast, assigning requests to EU-SOUTH-1 Spot instances would instead reduce the hourly rate by 4x, at the price of a 10% average frequency of disruption. Thus, the question is: *Which solution is best? Are there any preferable middle-ground solutions between these extremes?* Answering these questions before solving the optimization problem requires quantifying the relative preferences between the two objectives and formalizing them into a single-objective formulation. Instead, adopting a MO formulation allows to derive a set of "locally optimal" trade-offs, and decide which to deploy a posteriori.

Finally, the third objective minimizes the average frequency of replica interruption. Recall that Spot instances are the cheapest compute instances available, but their offered capacity can be subject to interruptions. Thus, if one or more K8s replicas are allocated to Spot instances, these might experience a service interruption if the cloud provider must request back the capacity to, e.g., allocate more ON-DEMAND or RESERVED instances for other clients. For this reason, the objective is to minimize the average interruption frequency relative to the total number of allocated replicas, as follows:

$$\min f_3 = \min \frac{1}{|D|} \sum_{i \in D} \sum_{j \in I_s} \sum_{k \in \mathrm{DC}} \sum_{r \in R_i} \sum_{t \in T} \frac{z^t_{i,j,k,r} f_{j,k}}{r_i} \tag{5.13}$$

With arguments similar to the previous discussion, Tables 5.13 and 5.14 show that minimizing the average interruption frequency is in competition with both latency and cost minimization. In other words, the Spot instances that offer the lower interruption frequency are not necessarily the ones that offer both the lowest latency and costs.

By jointly considering these three objective functions, we formulate our optimization problem as a MO-ILP. Specifically, we pre-compute the set D_t for each time-slot, containing the indexes of all scheduling and resource allocation assignments (i.e., starting time-slot, instance, and DC assignment) that are "active" (i.e., they must occupy CPU and RAM resources) at time-slot t. The MO-ILP formulation, comprising all the necessary constraints, can be expressed as follows:

$$\min\ (f_1, f_2, f_3) \tag{5.14}$$

$$\sum_{\substack{(i,t')\in D_t,\\ r\in R_i}} x^{t'}_{i,j,k,r} c_i \leq \begin{cases} r_{j,k} C_j, & \forall j \in I_r, k \in \mathrm{DC}, t \in T \\ o^t_{j,k} C_j, & \forall j \in I_o, k \in \mathrm{DC}, t \in T \\ s^t_{j,k} C_j, & \forall j \in I_s, k \in \mathrm{DC}, t \in T \end{cases} \tag{5.15}$$

$$\sum_{\substack{(i,t')\in D_t,\\ r\in R_i}} x^{t'}_{i,j,k,r} m_i \leq \begin{cases} r_{j,k} M_j, & \forall j \in I_r, k \in \mathrm{DC}, t \in T \\ o^t_{j,k} M_j, & \forall j \in I_o, k \in \mathrm{DC}, t \in T \\ s^t_{j,k} M_j, & \forall j \in I_s, k \in \mathrm{DC}, t \in T \end{cases} \tag{5.16}$$

$$\sum_{j\in I} \sum_{k\in \mathrm{DC}} \sum_{t\in T} x^t_{i,j,k,r} = 1, \qquad \forall i \in D, r \in R_i \tag{5.17}$$

$$x^t_{i,j,k,r} \leq a^t_i, \quad \forall i \in D, j \in I, k \in \mathrm{DC}, r \in R_i, t \in T \tag{5.18}$$

$$\sum_{t\in T} a^t_i = 1, \qquad \forall i \in D \tag{5.19}$$

$$d_{i,k} \geq \sum_{t\in T, j\in I} x^t_{i,j,k,r}, \qquad \forall i \in D, k \in \mathrm{DC}, r \in R_i \tag{5.20}$$

$$L_i \geq d_{i,k} l_{i,k}, \qquad \forall i \in D, k \in \mathrm{DC} \tag{5.21}$$

$$r_{j,k}, o^t_{j,k}, s^t_{j,k}, a^t_i, x^t_{i,j,k,r}, d_{i,k} \in \{0, 1\} \tag{5.22}$$

$$L_i \geq 0 \tag{5.23}$$

Constraints (5.15) and (5.16) impose the CPU and RAM constraints, respectively, for Reserved, On-Demand, and Spot instances. Constraints (5.17) impose that each request must be allocated its demanded amount of replicas, which can distributed across every available instance type and DC. Note that, while multiple replicas can be split across multiple instances and DCs, a single replica must be integrally allocated to a single instance and a single DC. Constraints (5.18) and (5.19) ensure that only one starting time can be chosen for scheduling each service request. Constraints (5.20) and (5.21) set the maximum delay experienced by each allocated request. Constraints (5.22) and (5.23) impose the variable domains. Fig. 5.14 shows an illustrative feasible solution for this problem.

This problem, along with similar problems in the context of packing and scheduling, is NP-Hard. Specifically, it can be considered a variant of the Temporal Multidi-

mensional Bin Packing Problem, a popular ILP model for representing resource allocation problems in cloud computing environments [222]–[225]. In this problem, differently from the literature, the bins (i.e., the compute clusters) are not assumed to be identical and have different capacities, pricing models, network latencies, and frequencies of service interruption depending on the bin type. All of these performance metrics play a role in their respective objective function.

The optimal solution to this MO-ILP is the complete PF, i.e., in a set of solutions covering all possible "optimal" trade-offs between the three objectives. Enumerating the PF is computationally challenging for large-scale instances, since it would require solving to optimality hundreds, or even thousands of single-objective ILPs [226].

As a reference scenario, a large-scale instance similar in size to those reported in the literature [227], [228] is considered, consisting of 50 requests with replicas ranging from 1 to 6 and a duration ranging from 10 to 50 time-slots (hours), over a scheduling period of 100 time-slots (hours). Each request is characterized by a per-replica required number of vCPUs (number of virtual cores) and RAM, a duration, i.e., the time the deployed replicas must remain active, and a requested number of replicas. To model round-trip latency, the assumption is that requests originate from a location close to one of the AWS EC2 DCs.

To realistically model the cloud instance specifications alongside their pricing models, *t4g AWS EC2* computing instances have been selected for the configurations 2XLARGE, LARGE, and SMALL to specify different capacities. For each of these computing instances, the hardware specifications and pricing details for the respective On-Demand, Reserved, and Spot rental options are illustrated in Table 5.13. Specifically, the scenario includes six EC2 DCs, located in different geographical regions: EU-SOUTH-1, EU-WEST-2, EU-EAST-1, US-WEST-1, AP-NORTHEAST-3, and AP-SOUTH-1. Furthermore, for Spot instances, the interrupt frequency is reported, which is the maximum frequency at which instances' interruption occurred according to Amazon's official reports [229]. Finally, the latency between the different EC2 DCs is modeled according to the Cloud Ping website [176], which gathers the average latency, i.e., the ping, collected over a one-month period between each pair of EC2 DC considered for this evaluation.

To numerically assess the quality of the approximated PFs, the following two performance metrics are considered:

- **Hypervolume metric**. Measures the volume of the PFs relative to a fixed reference point in the objective space, e.g., the worst possible values that the objective functions can take. A higher hypervolume metric signals a better approximation of the true PF.

- **Sparsity**. Measures the resolution of the approximated PF. Intuitively, a higher resolution (i.e., a large number of solutions in the PF) is more desirable, since it provides a greater degree of flexibility to the orchestrator. Sparsity is defined as

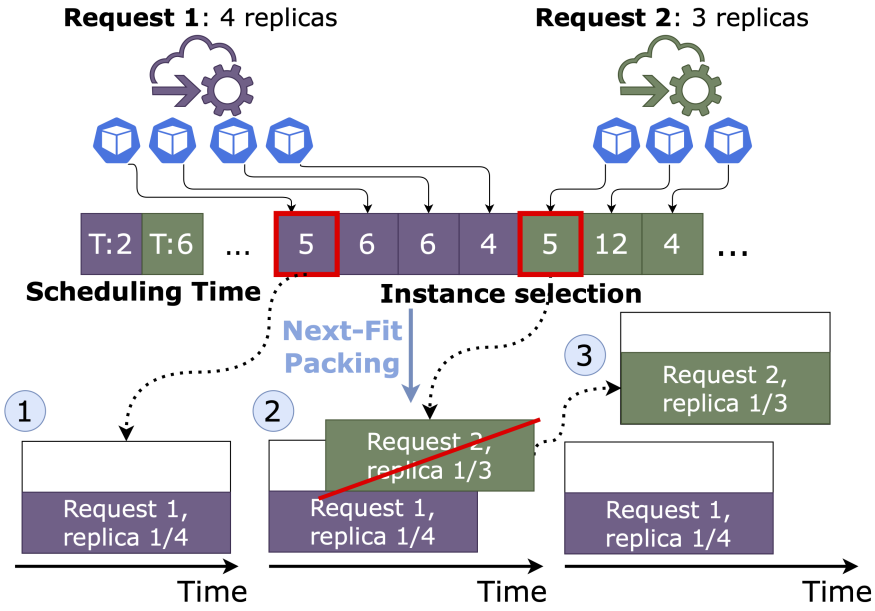


Figure 5.15: Scheduling and Resource Allocation decisions taken with MOEA. The encoding vector is divided into two parts: the first decides the scheduling for each request, and the second assigns each replica to a proper instance type (DC, type, size), encoded as an integer. Then, instances are created according to a next-fit algorithm: replicas are packed sequentially, and new instances are opened as soon as the capacity is exceeded.

[230]:

$$S(\mathcal{F}) = \frac{1}{|\mathcal{F}-1|} \sum_{j=1}^{m} \sum_{i=1}^{|\mathcal{F}-1|} \left(\tilde{\mathcal{F}}_j(i) - \tilde{\mathcal{F}}_j(i+1) \right)^2 \tag{5.24}$$

where m is the number of objectives, $\mathcal{F}$ is the PF approximation, and $\tilde{\mathcal{F}}_j(i)$ is the value of the j-th objective of the i-th solution, sorted by objective value. Lower sparsity implies a higher resolution of the approximated PF.

5.3.1 Multi-Objective Evolutionary Algorithms Encoding and Evaluation

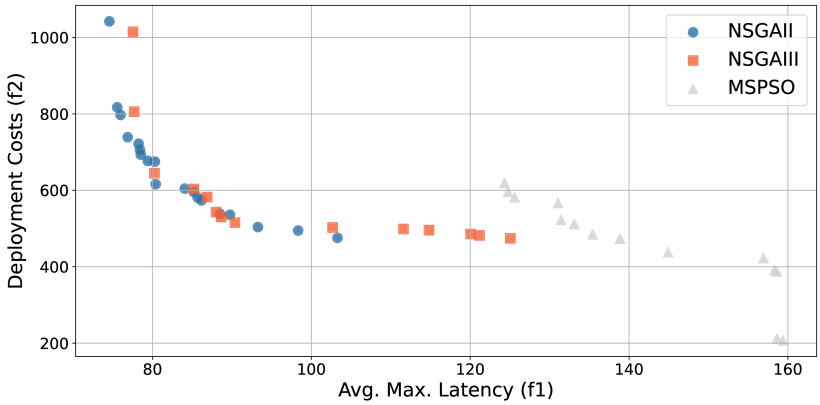
MOEAs require encoding a feasible solution of the optimization problem in a vector representation. Then, operators such as mutation and crossover are iteratively applied to a population of multiple vector representations to efficiently explore the solution space [150], [152]. The encoding of the scheduling and resource allocation problem is designed through an integer vectorized representation illustrated in Fig. 5.15, which shows both the scheduling and the resource allocation phases. The encoding vector is divided into two main parts: the first contains one element for each request to map the respective scheduling decisions, i.e., when to schedule the request in the time horizon, while

the second part contains one element for each replica. Specifically, the integer value of the element encodes a specific combination of the EC2 DC, the instance size (small, large, or 2xlarge), and the instance type: On-Demand, Reserved, or Spot instance.

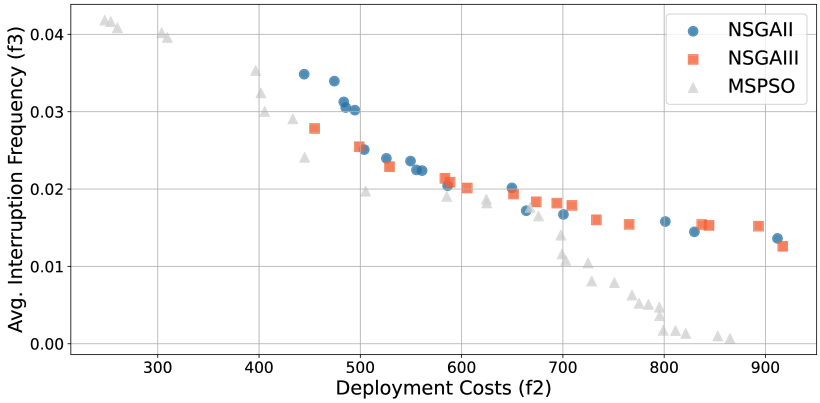
Decoding (i.e., converting the encoded vector into a feasible solution) is performed according to a classical Next-Fit Bin Packing algorithm, as illustrated in Fig. 5.15. Specifically, three different possibilities can be identified: i) if there are no active instances for a specific combination, a new instance of the selected type is activated, and the replica is allocated in it; ii) if there exists a running instance for the selected combination, and it has enough CPU and RAM capacity to accommodate the replica, and iii) if there exists a running instance of the selected combination, but it has not enough resources to accommodate the new replica, a new instance of the same time is activated and the replica is allocated in it. All the experiments presented have 50.000 iterations as the termination condition, after which no significant improvements in solution quality have been observed. On average, across 100 instances, the experiments took 24 seconds for NSGA-II and NSGA-III and about 110 seconds for MPSO.

Figures 5.16a and 5.16b illustrate the PF for bi-objective versions of the problem. Firstly, when optimizing latency and cost (Figure 5.16a), the NSGA algorithms and the MPSO converge on completely different solutions in the objective space. Specifically, while NSGA-II and NSGA-III favor solutions with higher costs and lower latencies, MPSO favors the opposite. Still, NSGA-II and NSGA-III obtain more practically useful solutions, in correspondence with the "elbow" of the PF, which signals a point of diminishing returns for either objective. In particular, NSGA-II provides the "best" PF approximation since it densely covers the elbow point with multiple solutions, providing the orchestrator with a fine decision-making granularity. In contrast, when optimizing cost and availability (Figure 5.16b, the MPSO produces a PF that strongly dominates NSGA-II and NSGA-III, providing a well-spaced set of nondominated solutions with no discernible elbow point signaling diminishing returns for either objective. The numerical validation of the qualitative comparisons above with the quantitative performance metrics is illustrated in Table 5.15. Since the algorithms run relatively fast and might focus on completely different parts of the solution space, the recommendation is to run them as an ensemble and combining the resulting PFs. Overall, these results highlight the benefits of a MO approach, producing tens of possible solutions (instead of just one), and providing an explicit way to identify the "ideal" trade-off regions, such as the elbow point in Fig. 5.16a.

Figure 5.17 illustrates the approximated PFs for the complete three-objective problem. The first observation is that the PFs for each algorithm are quite rich, comprising several tens of solutions (150 for NSGA-II, 112 for NSGA-III, and 100 for MPSO). This is because the flexibility provided by orchestrating single microservices enables a fine decision-making granularity, i.e., the balance between the three objectives can be slightly tilted by moving one or a few replicas from one cluster to another. From a practical perspective, this allows MO algorithms to produce a large set of nondomi-



(a)



(b)

Figure 5.16: PFs found by NSGA-II, NSGA-III, and MPSO considering the Deployment Costs and Latency as problem objectives (5.16a) and Deployment Costs and the Avg. Interruption Frequency as problem objectives (5.16b).

nated solutions, providing ample choices to the orchestrator. Although visually harder to discern compared to the 2D case, the quantitative performance metrics in Table 5.15 show that NSGA-III is the best-performing algorithm both in terms of both solution quality (measured by the Hypervolume metric) and resolution of the PF (measured by the Sparsity metric), with MPSO coming second. Indeed, similarly to what was observed in the bi-objective case, NSGA-III and MPSO attained convergence in different sections of the solution space. Specifically, NSGA-III discovers solutions with a few percent of the interruption frequency while trading off cost and latency, while MPSO discovers solutions with a low interruption frequency but higher costs. As before, considering this complex and multifaceted objective function landscape and the different metaheuristics exploration strategies, the recommendation is to run all of these algorithms as an ensemble to maximize the quality of the resulting PF and the number of

Table 5.15: Summary of the MO Performance Metrics

Objectives	Hypervolume	$S(\mathcal{F})$
Avg. Max. Latency (f1) & Deployment Costs (f2)	NSGA-II : **64264** NSGA-III: 62477 MPSO: 38293	NSGA-II : 3452 NSGA-III: 5730 MPSO: **2893**
Deployment Costs (f2) & Avg. Interruption Frequency (f3)	NSGA-II : 15.18 NSGA-III: 15.34 MPSO: **21.47**	NSGA-II : 1634 NSGA-III: 1276 MPSO: **887**
Avg. Max. Latency (f1) & Deployment Costs (f2) & Avg. Interruption Frequency (f3)	NSGA-II : 3694 NSGA-III: **4048** MPSO: 3930	NSGA-II : 203.29 NSGA-III: **123.79** MPSO: 166.77

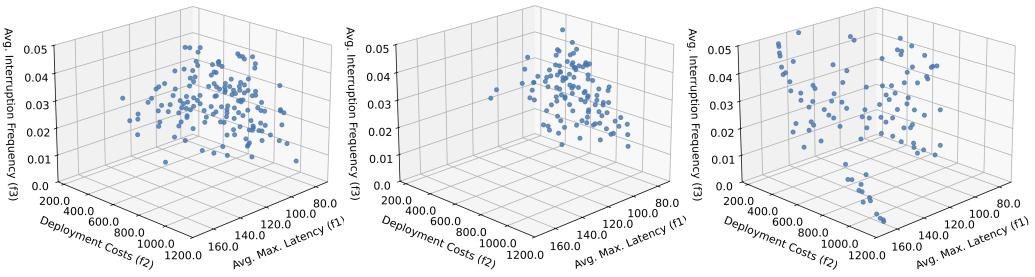


Figure 5.17: PFs found by NSGA-II, NSGA-III, and MPSO, considering total deployment costs, avg. request maximum latency and avg. interruption frequency.

available nondominated solutions.

5.4 Chapter Summary

Scheduling and resource allocation in microservice replicas across the CC is a fundamental and challenging optimization problem. This chapter investigated efficient multi-cluster orchestration strategies focused on the well-known K8s platform and in recent trends as RL, metaheuristics, and MOO approaches. It emphasized the need for extensive research in network management for multi-cluster orchestration to tackle interoperability and security challenges in distributed CC environments. Ongoing research, interdisciplinary collaborations, and standardized best practices are essential elements in the journey towards more efficient and reliable CC infrastructures. Section 5.1 introduced an overview of K8s principal aspects, discussing their dynamics and functionalities in the K8s ecosystem. Section 5.2 focused on the efficient scheduling of microser-

vices in a multi-cluster scenario. Specifically, in 5.2.1, an RL-based approach inspired on the OpenAI Gym library has been proposed to handle efficient multi-cluster orchestration in the well-known K8s platform. The evaluation considers two opposing strategies that show the feasibility of RL for the multi-cluster orchestration problem addressed in the paper. Results show that generalization can be achieved by incorporating the DS neural network in typical RL algorithms, achieving higher performance for scenarios 32 times higher than the trained one. Furthermore, 5.2.3 considers a MO reward function that demonstrates the feasibility of RL for the multi-cluster orchestration problem addressed in the paper. Results show that scalable and generalizable policies can be attainable by incorporating the DS neural network in typical RL algorithms, achieving high performance for scenarios higher than the trained one compared to heuristic baselines, without the need to retrain these algorithms. DS-based algorithms achieve minimal rejection rates (as low as 0.002%, 90x less than the baseline Karmada scheduler). Cost-aware strategies result in lower deployment costs (2.5 units), and latency-aware functions achieve lower latency (268–290 ms), improving by 1.5x and 1.3x, respectively, over the best-performing baselines. *HephaestusForge* framework is available open-source, allowing researchers to evaluate placement policies and potentially guide the development of additional scheduling algorithms.

Finally, Section 5.3 explored a novel MO formulation simultaneously accounting for instance pricing, latency, and interruption frequency. Conventional methods, which often combine these objectives into a single linear equation, fail to capture the complex nature of the trade-offs involved. Instead, this section proposed the MO problem by computing an approximated PF via custom metaheuristics. This allows decision-makers, such as network managers, to evaluate the spectrum of trade-offs and select the most appropriate strategy for their specific operational needs. Thanks to the flexibility provided by orchestrating individual replicas, the resulting PFs provide several tens of potential solutions (instead of just one, as with conventional approaches), with relative performance and qualitative trade-off regions that can be of major interest. Implemented metaheuristics are computationally fast for large-scale instances, and converge in distinct regions of the objective space, highlighting the benefit of running them as an ensemble and then combining the resulting PFs.

Chapter 6

Kubernetes Digital Twins for Service Orchestration

In the ongoing and ever-accelerating process of network softwarization, researchers are turning their attention towards *Digital Twins* [231]. The DT concept has emerged in industry 4.0 as a high-accuracy virtual representation and software companion for physical assets related to applications ranging from maintenance [52] to process and network optimization [53]–[55]. However, more recently, the term has also been applied to software platforms and digital assets. Along that way, some authors have already proposed DTs for mimicking and configuring/interacting with either networks, applications, or both [56]. In particular, DTs can address a wide range of situations by leveraging what-if scenario analysis and employing different mechanisms, spanning from performance optimization to chaos engineering [57]–[59].

Within this research avenue, a relatively recent development is the consideration of large-scale applications. Nowadays, large-scale applications are deployed on complex hybrid cloud scenarios, with many different public and private cloud environments and dynamic workloads, which present several challenges from the perspective of identifying optimal deployment configurations [57]. In practice, it is very hard to determine the optimal configuration of computational and network resources for a large-scale application before their actual deployment. This task is even more complicated by the adoption of sophisticated orchestration platforms, such as K8s. As previously discussed, K8s is becoming the de-facto solution for service management and orchestration, and it is increasingly proposed as a platform for a wide range of applications: from NFV implementation [60] to the tactical edge domain [61]. While this presents several advantages from the service provider perspective, it makes it even more difficult to assess upfront the proper configuration of a large-scale deployment, because its accurate evaluation must consider aspects that go well beyond the provisioning of resources, such as the number of VMs to rent, their prices, etc., and needs to evaluate the runtime impact of K8s control loops on the application behavior.

To address the above challenges, this chapter claims the need to design novel DT solutions purposely implemented by considering K8s and the requirements of K8s-based applications. Those solutions would allow us to accurately capture the state of an existing K8s-based IT application deployment through a virtual object with a smaller footprint and to be able to run what-if scenario analysis much faster than on a physical testbed. This would allow for efficient and parallel evaluation of the behavior of the DT using different configuration parameters or even modified components, with many relevant applications ranging from design feedback to resource optimization and CE.

6.1 The case for a Kubernetes Digital Twin

There is no clear definition of the DT concept [232], and several have been proposed over the years [52]. However, most of the definitions seem to agree that a DT is a system composed of 3 elements: a physical object (or system), a high-accuracy virtual representation of it (often obtained by adopting sophisticated simulation solutions), and an active bidirectional link between those elements, that allows aligning the state of the physical object and its virtual representation. Extra components for performance evaluation and optimization, or more generally decision-making, are often present. In the literature, the term DT is, rather ambiguously, used to define the virtual representation element, the entire system, or both. In a DT, the link between the physical and virtual elements is bidirectional: it goes *from the real system to its virtual representation*, thus allowing the latter to keep track of state changes in the real system, and *from the virtual representation to the real system*, thus allowing to put in place changes in the system configuration that were explored in the DT and judged more satisfactory than the previous ones. It is important to note that while the link between the physical and virtual elements is bidirectional, the virtual element can be used to run disconnected or "offline" what-if scenario analyses. During this process, the virtual element explores several configurations without altering the physical element. Only when a satisfactory configuration is found will it be translated to the physical element.

In the network and service management domain, researchers and practitioners deal with large-scale and very complex networks and applications that exhibit quite a dynamic behavior, and they constantly explore new methodologies and tools that help them in the struggle to maintain and optimize their systems continuously [233]. For this reason, they are increasingly adopting sophisticated orchestration solutions such as K8s, which provides functional management tools, including network management, automated deployment, and autoscaling assistance that simplify the management of complex and large-scale applications. K8s adopts a declarative approach to application deployment, implementing a wide range of complex control loops that continuously monitor applications and modify their deployments to match the desired state. K8s adopters are required to provide a detailed description of their application in terms of

container specifications, deployment configurations, and expected Key Performance Indicators. However, while K8s provides valuable functions from the operation automation perspective, including dynamic behaviors such as autoscaling and software update management, the large number of control loops it implements significantly enlarges the already vast space of configuration parameters and policies to consider for management purposes.

This domain would significantly benefit from DT approaches, extending and redefining the concept as shown in Fig. 6.1. In this case, the physical system is itself a digital object: a large-scale K8s-based deployment built on top of many microservices, which in turn are executed in a federation of K8s clusters, typically with complex configurations that control orchestration and automation behavior at several levels: cluster, application, and software component. This represents a very intricate system with significant dynamic aspects due to varying workloads, complex service deployments, heterogeneous network fabric, etc., whose accurate reenactment requires capturing the system behavior at the application and platform levels. Creating a virtual representation of this system presents critical challenges at the simulation level (both from the performance and accuracy perspectives), as well as at the performance assessment and optimization/decision-making levels. In fact, optimization criteria are typically defined at the business level, thus resulting in an additional layer of complexity to the performance assessment and significantly complicating and enlarging the space of possible configurations to explore for decision-making.

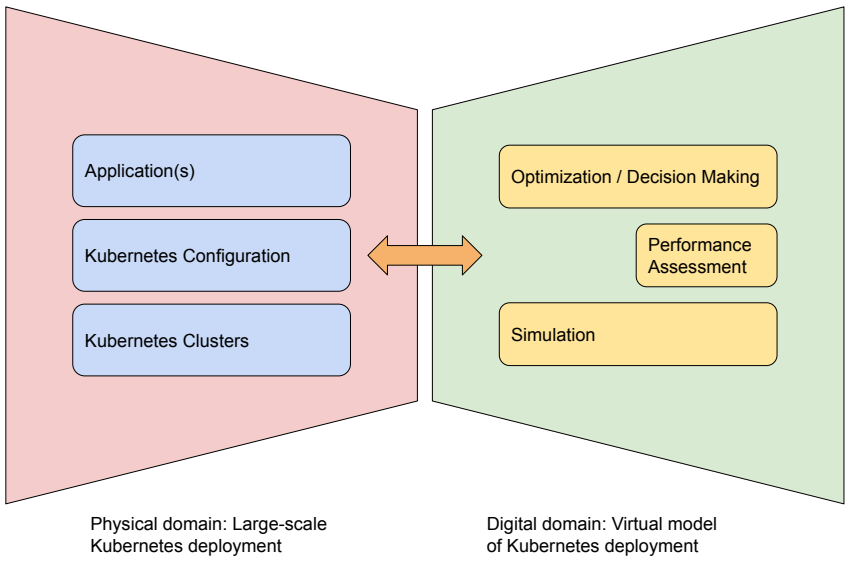


Figure 6.1: The Digital Twin concept redefined to consider Kubernetes-based software deployment.

Developing DTs of K8s-based deployments calls for innovative solutions capable of addressing the problems of accurately reenacting large-scale K8s-based software deployment, evaluating their behavior under different operating conditions, identifying potentially more optimal configurations, and reifying configuration changes by interacting with the control APIs of the physical K8s cluster(s) considered. However, effective development of DTs requires sophisticated monitoring tools that can capture the behavior of the K8s deployment, such as computing and network resource monitoring, performance estimation, and so on. The collected metrics will be used to define the DT and manage the K8s deployment.

For this reason, we developed KT as a valuable framework to define and reenact DTs of K8s-based software deployment. KT is a comprehensive tool that can study the behavior of K8s deployment using a DT approach to run what-if scenario analyses. This tool allows its adopters to identify optimized deployment configurations, evaluate various scheduling policies, experiment with different load-balancing algorithms, and finally translate these configurations and policies into K8s specifications.

6.2 KubeTwin: A Digital Twin framework for Kubernetes deployments at scale

KT is a framework that allows the creation of DTs of K8s-based software deployment and the evaluation of their performance in a deployment scenario of interest[1] [234]. To execute the DT, KT reenacts the behavior of a K8s deployment at a very fine-grained level, simulating each service request as it travels. Particular attention was dedicated to the definition of Business Process Execution Language-like workflows on top of service components, to the implementation of control loops for service component replication, and to the accurate modeling of network communication latency. This is to provide a higher level of accuracy when compared to the use of simpler regression models, which cannot simulate complex and dynamic scenarios, and to provide realistic insights regarding resource utilization and response times under changing conditions. As K8s, KT logically divides sets of computing nodes into clusters. Each cluster is identified by its name, type, location, and the number of computing nodes. In addition, a cluster defines a configurable amount of computing resources, e.g., CPU or GPU cores, to specify the performance of its computing nodes. We adopt this design choice to enable the modeling of different types of clusters that KT can use to deploy the application components. In particular, with KT, we can define two types of computing nodes: edge nodes, e.g., MEC servers hosted in small-size data centers close to the end-user

[1] D. Borsatti, W. Cerroni, L. Foschini, et al., "Kubetwin: A digital twin framework for kubernetes deployments at scale," IEEE Transactions on Network and Service Management, vol. 21, no. 4, pp. 3889–3903, 2024. doi: 10.1109/TNSM.2024.3405175

premises, and medium- and large-size data centers located at cloud facilities. This solution enables the reenacting of complex, heterogeneous, multi-cluster deployments. For example, it is conceivable that shortly a service provider could distribute an application by exploiting both MEC and cloud computing resources. In this case, nodes available at cloud facilities would likely provide much more computational resources than MEC nodes. KT applications are defined using a declarative description, which is semantically equivalent to K8s, adopting a flexible approach that allows its users to specify different kinds of applications, such as multi-tier web applications or complex money transfer management systems [57]. To this end, KT users will describe complex services as the composition of multiple microservices that interact together. In addition, KT groups user requests into workflows, which specify the coordinated execution of a subset of microservices.

Regarding the execution model, a DT created with KT could run in two operational modes: "offline" and "online". In the "offline" mode, KT functions independently from the live K8s environment, utilizing historical data and predefined scenarios to perform risk-free what-if scenario analyses, thus allowing service providers to explore potential outcomes and optimizations without impacting the actual deployment. We believe that the "offline mode" is a resource-efficient approach, ideal for training, planning, and testing changes in a simulated environment. On the other hand, when running in the "online" mode, KT operates in parallel with the live K8s system, thus implementing a bidirectional real-time link between the physical and virtual elements of the DT. In this mode, it continuously integrates and processes real-time data, enabling dynamic monitoring and fine-grained tuning of the K8s deployment configurations. Both operational modes can bring valuable insights to optimize the behavior of a container-based application deployed on K8s, the "offline" mode is the best choice to identify the best configuration before deploying the application at scale.

KT is written in Ruby and distributed open-source (MIT license) on GitHub[2]. Ruby is a high-level, object-oriented programming language well known for its expressiveness and ease of use. This makes it a good choice for developing complex systems like discrete event simulators. In fact, the dynamic nature of Ruby allows for code to be written in a more natural and readable way, reducing development time and improving code maintainability. KT is implemented as a single process discrete event simulator to reenact the behavior of a service managed on top of the K8s orchestration platform. KT users need to create a deployment file describing the service to provide, its configuration, the available clusters to allocate the service components, and the distribution of user requests.

[2] `https://github.com/DSG-UniFE/KubeTwin`

6.2.1 KubeTwin Components

Figure 6.2 illustrates the interaction between KT components, which are designed to implement a realistic KT of the K8s framework. Each element is developed with the idea of making it as compliant as possible with an actual implementation of K8s. Specifically, KT implements these components to reenact the K8s functionalities accurately, such as name resolution, load balancing, scheduling, and so on.

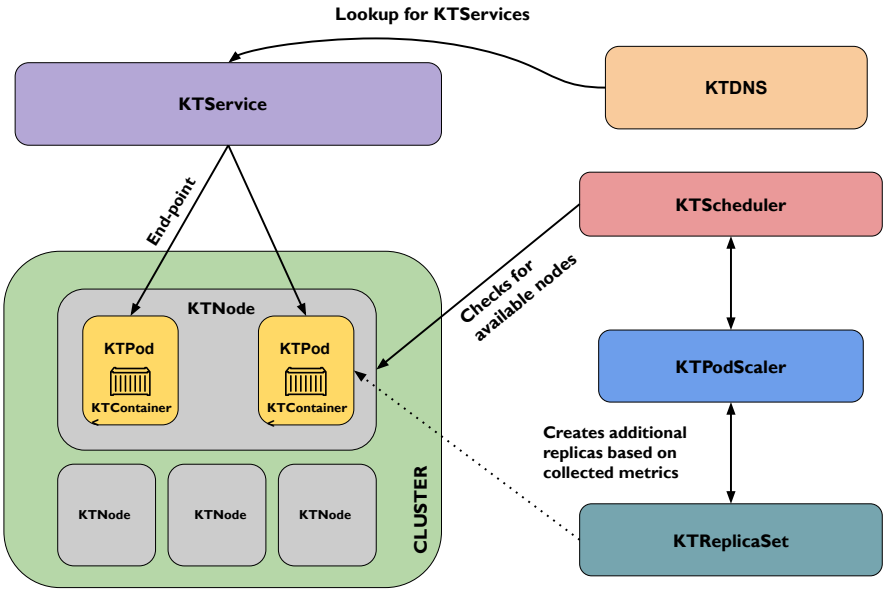


Figure 6.2: KT Architecture and interactions between components.

First, a KTService is used to represent a K8s Service. KT implements KTService as a list of associated pods that match a given selector, i.e., the KTService name. Each list element represents an "endpoint", a symbolic link that maps a KTService with an associated pod. In addition, each KTService must define a load balancing policy to distribute the load of processing requests among associated pods. This load balancing policy could be one of the default ones implemented by K8s or even user-defined rules that extend the default behavior, e.g., location-based load balancing. Currently, KT implements a load balancing policy that mimics the default one implemented via iptables in Kubernetes. In addition, KT can also assign incoming requests to different pods in a round-robin fashion. Other policies can be easily defined using the KT framework. For instance, to minimize communication delay, KT users may want to specify a latency-based load balancing policy that assigns incoming requests to pods running on

computing nodes near the requesters.

The KTDNS, which implements the naming resolution and lookup functionality within KT. Specifically, KT registers all KTServices in KTDNS using a label as a unique identifier for a KTService. Once registered, KT components can query the KTDNS to retrieve information about active KTServices. The KTReplicaSet resembles the functionality of the K8s ReplicaSet component. KT users need to define a KTReplicaSet for each KTService to specify the number of pods associated with the KTService. Precisely, KT models a KTReplicaSet using two parameters: a selector to identify the corresponding KTService and the number of associated replicas. At the beginning of the simulation, each KTReplicaSet instantiates the specified number of replicas, then it continuously monitors their status to activate new ones when necessary, e.g., following the crash of a pod.

Regarding the models for the computation elements, we define a container as a single unit of execution. Each container implements a single software component that describes the amount of CPU and memory required. Using a recurrent choice in the scientific literature, which also well suits the simulative approach [235], we use a G/G/n/First In First Out (FIFO) queuing model to reenact the process of serving requests at the software component level. More specifically, KT users can configure the level of parallelism, the maximum queue size, and the request service times associated with a specific software component - in the latter case, by defining a random variable from which KT will sample the service times. KT provides a wide range of well-known distributions, from exponential to log-normal to Pareto-family ones, but also allows the use of empirical distributions obtained from real-life measurements. This design choice gives KT users significant freedom to explore model accuracy against complexity and bias against variance trade-offs, within a conceptual framework that is well-understood and relatively easy to work with.

It is also assumed that a pod hosts a single container. This design choice simplifies the design of KT while still allowing us to accurately model the overwhelming majority of K8s applications, which adopt the "one container per pod" policy (usually considered a best practice). However, future versions of KT plan to support multi-container pods to enable the accurate reenactment of the small share of applications that leverage pods with multiple affine containers (typically a main container and one or more "sidecar" containers). The KTScheduler plays a relevant role within KT. It is responsible for managing computing resources according to a set of configurable policies, including the result of automated scaling procedures. Specifically, the KTScheduler selects the computing node where to activate a new pod considering its requirements, the status of available computing resources, and the configured scheduling policy [236]. A three-step procedure regulates the selection of a computing node. Firstly, KTScheduler filters the resources of the available clusters to retrieve a list of computing nodes with a residual computing capacity to fit the new pod. Secondly, the KTScheduler assigns a score to the filtered nodes following a configurable scheduling policy. Finally, KTScheduler

selects the node with the highest score as the candidate to deploy the pod. To this end, it is worth noting that the KTScheduler executes the filter-and-score procedure each time there is a request for a new pod.

KT users can tune the KTScheduler behavior by providing their own scoring policy, thus allowing them to experiment with custom resource schedulers for K8s in reproducible environments - a very hot research topic [237]. Interesting yet simple examples of scoring policies could be sorting the filtered nodes according to the highest available residual capacity, thus leading to relatively evenly distributed resource allocations that maximize the responsiveness of autoscaling processes, or the lowest renting prices to minimize the overall service provisioning costs. A different, slightly more complex, scoring policy could be to assign a higher priority to the nodes located in the proximity of users' premises. This could be a reasonable choice for MEC applications with low latency requirements. Finally, it is important to specify that while the configuration of these components might be slightly different from the one used in K8s, it is easy to translate KT-specific configuration into Kubernetes and vice versa. This translation effectively implements a bidirectional link between the DT running on the top of KT and the K8s deployment.

6.2.2 KubeTwin Automated Scaling

The components described in the previous Subection allow KT users to simulate a static application deployment. However, the power of K8s also resides in its automated scaling solutions. Among those, the HPA is a well-adopted solution that K8s can leverage to scale a deployment according to the current workload.

To reenact autoscaling behavior, KT provides the KTPodScaler component, as illustrated in Fig. 6.3. KTPodScaler is implemented as a periodic control loop to check the current performance of the associated KTReplicaSet, at configurable time intervals. According to the HPA specification [238], KTPodScaler monitors the performance of the application by calculating the average processing time of the associated KTService, considering all instantiated replicas. If the current processing time $T_{\text{proc,current}}$ is higher than the expected value $T_{\text{proc,desired}}$ (i.e., the current number of replicas cannot handle the number of requests) the KTPodScaler will immediately try to activate new ones. To avoid under/over scaling of application components, KT users must specify the values for $maxReplica$ and $minReplica$ parameters, which set an upper and lower bound for the pod replicas associated with a KTReplicaSet.

More specifically, KTPodScaler calculates the number of replicas as follows:

$$N_{\text{replicas}} = \left\lceil N_{\text{replicas,current}} \times \frac{T_{\text{proc,current}}}{T_{\text{proc,desired}}} \right\rceil \tag{6.1}$$

where N_{replicas} should not exceed the specified $maxReplica$ parameter and be lower than the $minReplica$ parameter. To specify the $T_{\text{proc,desired}}$ parameter, KT users can

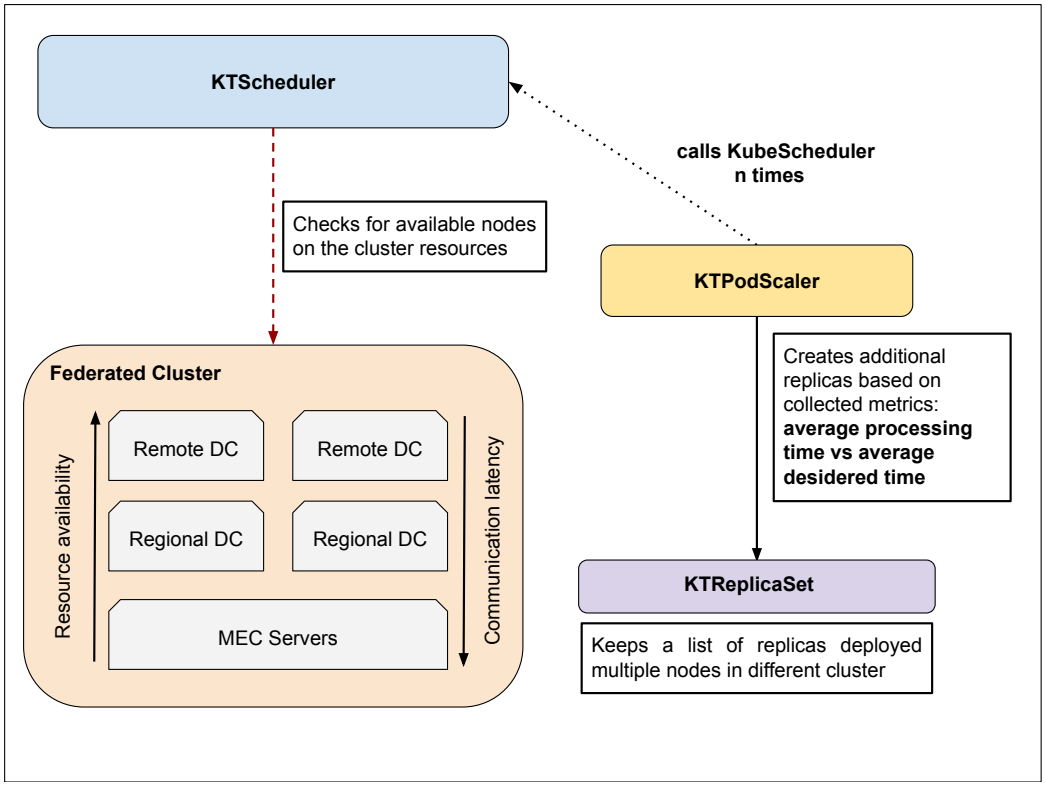


Figure 6.3: The main operations performed by KTPodScaler for allocating pods on the available computing nodes in the federated cluster.

set a tolerance range to indicate an interval within which the time to process a certain request type is acceptable. For example, KT users may decide to maintain the same number of replicas if $T_{\text{proc,current}}/T_{\text{proc,desired}} \in [0.9, 1.10]$, i.e., a 10% tolerance range.

To decide where to activate the new replicas, the KTPodScaler interacts with KTScheduler, which runs the filter-and-score procedure to select the computing nodes. Let us note that the KTPodScaler works in both ways, i.e., to increase or decrease N_{replicas} according to the current workload to avoid a waste of computing resources. It is worth noting that a KTPodScaler is KTService-specific. This means that KT users can choose whether to create a KTPodScaler for each KTReplicaSet of their application. For example, in a complex microservice application, only some software components may be associated with the auto-scaling feature, while others could not. Finally, the evaluation of the performance of application autoscaling within KT represents an important link from the virtual representation of a K8s application to its real-life counterpart. In fact, when KT identifies well-performing values (or value ranges) for KTPodScaler configuration parameters (i.e., an optimal number of replicas for a given component), it sends that information to the real K8s deployment to be used for priming the autoscaling config-

uration – allowing it to work in a proactive fashion or even with a predictive process.

6.2.3 KubeTwin Communication Model

Following the approach used in [57], the latency values between software components have been modeled using random variables. This approach is far more accurate than other approaches considering either static communication latency values or Euclidean distances between two distinct locations. Instead, the use of random variables enables the modeling of more realistic communication latency values, which could be sampled from historical data distributions or purposely defined distributions to test, for instance, the effect of increased latency values in communication links and so on. Specifically, these random variables are completely configurable by KT adopters, including the case of random distributions generated from latency values measured in real distributed computing environments, to allow a realistic reenactment of different use cases. This approach enables the specification of distinct latency models for pods running within the same cluster ("intra-cluster communications") and pods running in separate ones ("inter-cluster communications"). Therefore, it increases the realistic degree of the simulation and lets users specify complex deployment scenarios such as CC deployment, in which some of the software components are distributed at the edge of the network, while others are running in cloud computing facilities. Delving into implementation details, KT employs "locations" to identify different communication endpoints, i.e., data centers in distinct geographical locations. The latency between these locations is specified through a matrix notation with elements $RV(i, j)$. Each element $RV(i, j)$ is a random variable modeling the latency between locations i and j. Therefore, KT calculates the network delay for a given pair of locations i, j by sampling from the corresponding $RV(i, j)$. Finally, communication latency between two endpoints can be symmetric $RV(i, j) = RV(j, i)$ or asymmetric $RV(i, j) \neq RV(j, i)$. In the former case, the matrix would be symmetric, while in the latter one, the matrix would be a square matrix.

6.2.4 Characterization of Microservice Response Time in Kubernetes

DT approaches could represent a compelling step forward to approximate the behavior of K8s applications, thus allowing the evaluation of different configurations safely and quickly. In fact, DT approaches enable what-if scenario analysis [57], [239], thus implementing a faster (and parallelizable) process for the exploration of a larger number of configurations, and even allowing the rapid prototyping of custom K8s functions (e.g., autoscaling, scheduling). As a result, DTs could be very effective in speeding up the parameter identification process, as well as in significantly broadening its scope, with potentially significant cost savings [240], [241].

However, defining an accurate DT of a K8s application to fully exploit KT functionalities is a challenging task. Several variables need modeling, such as the average container processing time, the communication latency between the cluster's nodes, and internal K8s control loops. These parameters can be defined a priori or collected from historical data through statistical analysis. It is important to note that the quality of these parameters influences the ability of the DT to accurately reenact the behavior of its K8s application. Among these variables, modeling the service time of each microservice arguably represents the most critical step in the realization of the DT. In the current literature, there are not many consolidated solution to model the statistical distribution of the response time of K8s applications. A good statistical distribution of the service time is essential to have a simulation environment with a good approximation of the modeled application. Leveraging KT, it is possible to analyze the problem of parameter identification to create an accurate statistical description of a K8s application.

As a first solution, this subsection presents an innovative simulation-based inference procedure [242] that identifies the required configuration parameters. Specifically, the inference procedure explores the space of possible configuration parameters, comparing the metrics collected from a real-life K8s application and the outcome of a DT simulation with a set of sample parameters, and identifying the set Σ that minimizes the statistical difference between those two observations[3]. In other words, our procedure solves the following optimization problem:

$$argmin_{\Theta,\Sigma} f(x, y) \tag{6.2}$$

where x represents a set of observations taken from the K8s application and y is the observations taken from a simulation run using a model Θ with parameters Σ. The objective function $f(x, y)$ measures the statistical difference between the probability density functions of x and y, thus indicating how distant the observations generated from the DT are from those generated from the K8s application. Simulation-based inference represents a relatively computationally intensive process, and in this case (6.2) usually has to deal with a relatively large search space to explore. However, results obtained show that these approaches are far more accurate and robust than other approaches leveraging simplistic approximations, e.g., matching the mean processing times values of the target microservices. Although the expected observations x come from metrics collected from a K8s application, the observations y are generated by the DT with a model configuration $\Theta = m_1, m_2, \ldots, m_n$. Θ is a set containing n – where n is the number of microservices that compose the application – random variables m_i, describing the processing time of the corresponding microservices. As for the modeling of the random variable $m_i \in \Theta$, the simulation-based inference procedure provides several possibil-

[3] D. Borsatti, W. Cerroni, L. Foschini, et al., "Modeling digital twins of kubernetes-based applications," in 2023 IEEE Symposium on Computers and Communications (ISCC), 2023, pp. 219–224. doi: 10.1109/ISCC58397.2023.10217853.

ities to describe their distributions. Therefore, users can choose the distribution that could better fit the microservices processing times: a log-normal distribution, a Gaussian distribution, a Gaussian Mixture Model (GMM) with a configurable amount of components, and so on.

To solve problem (6.2), the optimizer looks for the configuration parameters $\sigma_i \in \Sigma$ of all random variables $m_i \in \Theta$ that allow the DT to produce observations y that are most statistically similar to x. This procedure allows the DT to predict the behavior of the K8s application with higher precision. There are several possible methods to measure statistical similarity, including calculating the Kolmogorov–Smirnov statistics of two observations x and y [243], their Wasserstein distance [244], or their Wilcoxon-Mann-Withey distance [245]. The outcome of the simulation-based inference procedure will be the service time configurations for Θ to be used for reenacting the DT model of the K8s application. The outcome of the simulation-based inference procedure will be the service time configurations for Θ to be used for reenacting the DT model of the K8s application. With regard to the problem complexity, it is worth noting that the type of distribution influences the solution of the optimization problem (6.2), i.e., increases or decreases the dimension of the solutions' search space. Specifically, finding a solution to the optimization problem (6.2) would require exploring a relatively large search space, which we can approximate using the following:

$$D \approx n \times \sum_{i=1}^{n} k_{m_i} \tag{6.3}$$

where n is the number of microservices and k_{m_i} the number of components chosen to describe the random variable m_i.

Concerning the optimization algorithm to implement the simulation-based inference procedure, the QPSO algorithm of the ruby-mhl [162] has been leveraged. The reason behind this metaheuristic approach lies in its ability to strike a good trade-off between convergence speed and to effectively explore large spaces in search of global minima [57]. An image recognition application has been chosen as a reference use case to be discussed. The image recognition service is an application that lets users identify objects in a picture, such as a photo captured by the user's phone or camera feeds' frames. When invoked, the application returns a message containing the names of the recognized objects to the user who requested the service. It is conceivable that this application would run on dedicated computing resources given its high computational requirements, allowing users to save the battery life of their equipment [246]. The image recognition app is a common example of an edge computing application, as it can process raw data produced by near cameras or Closed Circuit Television (CCTV) [247]. Processing images directly at the edge of the network is beneficial in reducing network latency and usage since it avoids uploading collected image frames to the cloud for processing.

The image recognition app is implemented as a chain of two different microservices, the first implementing image resizing and the second implementing an object recognition algorithm. Both microservices leverage the Flask Python framework to allow network communication via REST-ful APIs. The first chain microservice (MS1) takes the image and normalizes it, creating a representative array. Then MS1 sends this array to MS2, which classifies the image by leveraging a pre-trained model classifier. The final output of MS2 is the name of the object recognized from the input image. This information is then relayed to the user through MS1. Regarding orchestration, MS1 and MS2 have been encapsulated as two different containers allowing K8s to manage and orchestrate them. Then, a small-scale testbed composed of two VMs with four vCPU cores and 8GB of RAM each run a standard installation of K8s. To make the image recognition application available, we deploy the two microservices as two different containers running in separate pods. In addition, for each microservice, we define a Kubernetes NodePort service to handle service name resolution, load balancing functions, and exposing the application endpoints. Finally, a deployment for each microservice manually scales the number of replicas of each component.

The first step to define an accurate DT model for the image recognition application relates to the metrics collection, with a particular focus on the estimation of the processing time for MS1 and MS2 and the Time To Resolution (TTR), i.e., the total time to serve a generic request in a baseline scenario using the testbed deployment described above. The baseline configuration is obtained by analyzing the application behavior in a steady-state scenario, using one replica for each microservice and a workload of non-overlapping requests (i.e., no queue time). A request generator using the Python programming language collects these metrics, thanks to its functionalities to send requests with a configurable interarrival time to an HTTP endpoint. For the baseline scenario, the request generator sends 2000 requests with a rate of one request per second. With this configuration, the interarrival time between subsequent requests is much larger than the average service time (around 50ms), thus canceling the requests' queuing time. In fact, minimizing the effect of queuing times is essential to properly capture the behavior of service processing times and to make sure that the system under analysis is stable and operating in steady state conditions.

For each request, the end-to-end TTR has been collected along with the service time for MS1 and MS2 by collecting the time required to execute the instructions within each HTTP function from their logs. Then, a Simple Moving Average (SMA) with $n = 50$ computed the processing times for all observations of MS1 and MS2. The behavior obtained by these calculations shows that MS1 and MS2 appear as stationary processes since their moving averages do not have high variations. Specifically, the SMA of MS1 is around 10 ms, while the one for MS2 is around 30 ms. By assuming the stationarity, a G/M/1 queue model should describe well the service processing function of both microservices. Therefore, the simulation-based inference process described should be able to find a DT model that well fits the image recognition application. The first re-

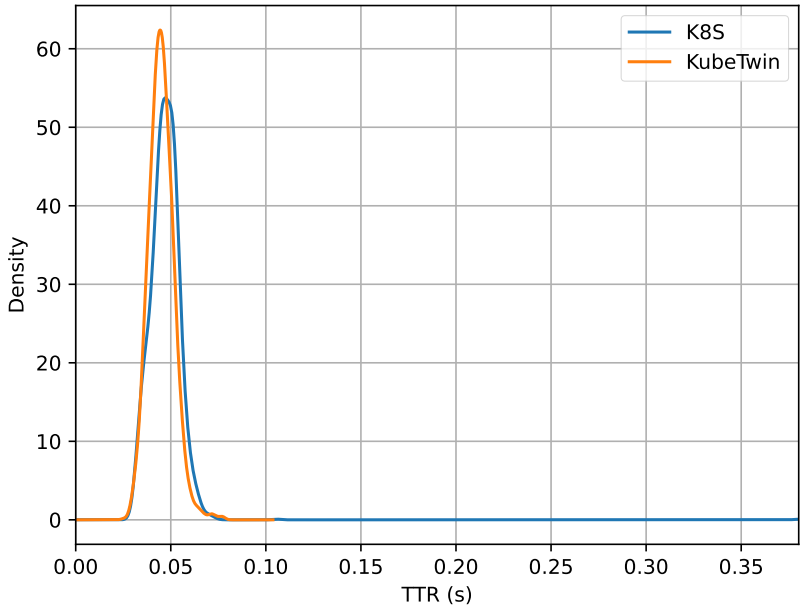


Figure 6.4: The results of the simulation-based inference process.

quirement for the fitting process is to define a KT configuration file containing the description of the two microservices, the available computing resources, and other K8s-related configurations. For these experiments, the processing time of MS1 and MS2 is fitted using two GMM random variables with three components. Each component has a weight, a mean, and a sigma parameter with fixed lower and upper bounds to help the convergence of the optimization algorithm. Concerning the QPSO parameters, the number of iterations was set to 200, the size of the swarm to 50, and the α contraction-expansion parameter to 0.75. At each iteration, the optimizer evaluated 50 different configurations, from which it selected the population's best until it reaches the maximum number of iterations. As a function to measure the statistical distance between the simulation outcome and the metrics, the Wasserstein distance implemented in the SciPy Python library was used.

The simulation-based inference process in Fig. 6.4, which shows the Probability Density Functions generated using i) the image recognition application's log (K8s) and ii) the best configuration found by the simulation-based inference process (KubeTwin). The proposed method is able to generate an accurate model for the DT that, when executed, can reflect the distribution of the end-to-end TTR values of the real image recognition application. Furthermore, the distributions illustrated in Fig. 6.4 have a long right tail, thus indicating some high TTR values peaks for the image recognition

application. These peaks are probably the results of naming resolution, waiting times of containers, and other situations in which occurrences and distributions are difficult to model. However, these peaks are limited in number (1 or 2), thus not changing the steady-state behavior of the K8s application. Overall, the results illustrated in Fig. 6.4 show that the proposed method can find an accurate DT model that mirrors the real K8s application under the baseline scenario. However, the DT model should still provide good accuracy even when operating in different working conditions (e.g., higher requests load). Therefore, to verify the accuracy of the DT model, an experimental campaign collected and compared the TTR values from 5000 requests for both the K8s application and the DT under different workloads. The idea is to show that the DT model can well reproduce the behavior of the real application in a modified scenario, thus evaluating whether the fitted model can still be a good representation of the image recognition application. In this regard, a Python application was implemented to send a total amount of requests at different Requests per Second (RPS) values. This is because even if there are several tools available that can execute a stress test of a web-based application (e.g., the Apache benchmark tool), none of them allows the user to specify a target RPS value. In fact, the capacity to stress an application using a target RPS mainly depends on the computing resources available on the machine generating the requests. For this validation, the end-to-end TTR values of all 5000 requests were collected for each experiment to compare and verify their distribution. Fig. 6.5 shows the Mean TTR and the 99th percentile values on the y axis that we collected from the execution of both the image recognition application (K8s) and its DT for all experiments (from 1 to 20 RPS) using the baseline deployment, one replica per microservice.

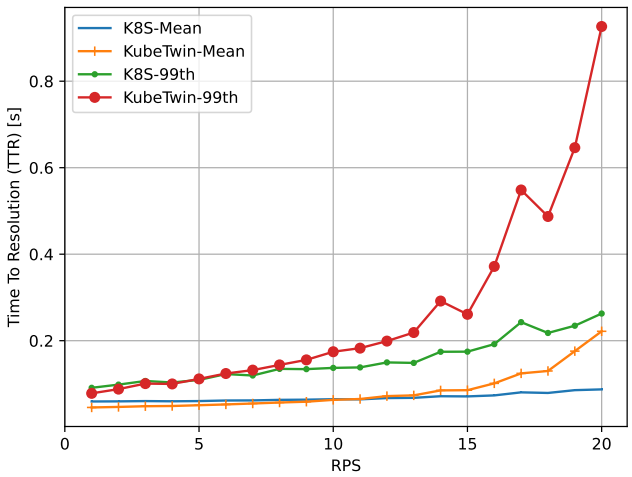


Figure 6.5: The average and the 99th percentile TTR values collected for 5000 requests with KubeTwin and the K8S under different workload RPS.

Observing Fig. 6.5, it is easy to verify how the mean TTR values computed by the DT model are a good approximation of the image recognition application until around 15 RPS. Then, starting from 16 RPS, the mean TTR values of the DT model diverge from the ones of the application. Considering the high variance of the results, analyzing only the mean TTR values could not be enough to understand the model performance. Therefore, Fig. 6.5 also shows the distribution of the 99th percentiles for the application and the DT, which can provide a better insight. Analyzing the 99th percentile values, the divergence between the simulation and the application is even more evident. The differences revealed that the DT model was probably overestimating the service time of the microservices at higher RPS. To prove this behavior, Fig. 6.6 gathers the logs from the microservices at different RPS . From these results, it is evident that the service time of each microservice decreases with the higher request rate. This speedup effect in the response time is probably caused by optimizations made at the CPU level (e.g., caching) that increase the performance of very active processes. Following this finding, the value obtained for RPS 20 has been used to re-fit the model with the same procedure. Then, the measurement campaign has been repeated and the results plotted in Fig. 6.7. As visible in the picture, the model is much more precise in forecasting the application response time, with a Mean Square Error decrease of approximately 0, 0013 for the model fitted at one (Table 6.1.)

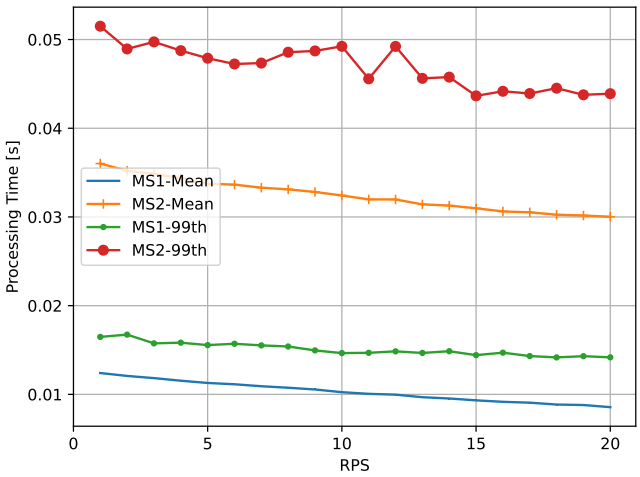


Figure 6.6: The average and the 99th percentile of the processing time for MS1 and MS2 from RPS 1 to RPS 20.

Table 6.1: MSE Values for the MTTR and the 99th percentile

Model	MSE - Mean	MSE - 99th
Baseline (1R)	0.001641	0.042060
RPS 20 (1R)	0.000364	0.001200
RPS 20 (3R)	0.010873	0.077834

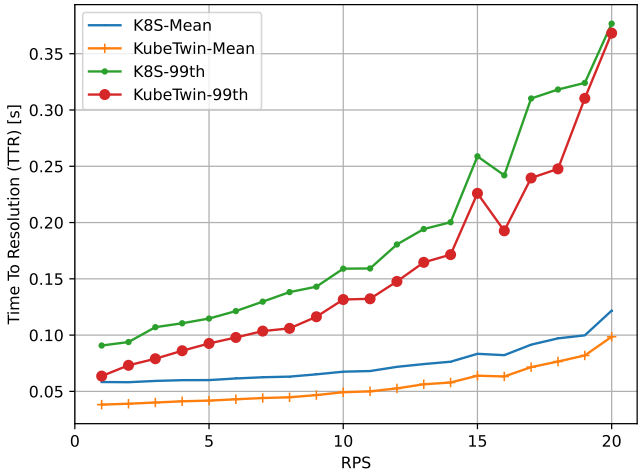


Figure 6.7: The average and the 99th percentile TTR values collected for 5000 requests using the model Fitted at RPS 20.

As a second validation, the DT model abilities to approximate the behavior of the K8s application when the deployment increases in size have been verified. More specifically, the number of replicas was set at 3 for each microservice, and the TTR logs from 5000 requests under different workloads from RPS 1 to RPS 40 were collected. At the same time, the DT model run using a configuration of 3 replicas to compare the results. These are visible in Fig. 6.8, which illustrates how KT can provide a good estimate of the real application up to 40 RPS. From then, the two systems diverge, and the 99th percentile of the DT model reports a higher value compared to the K8s application.

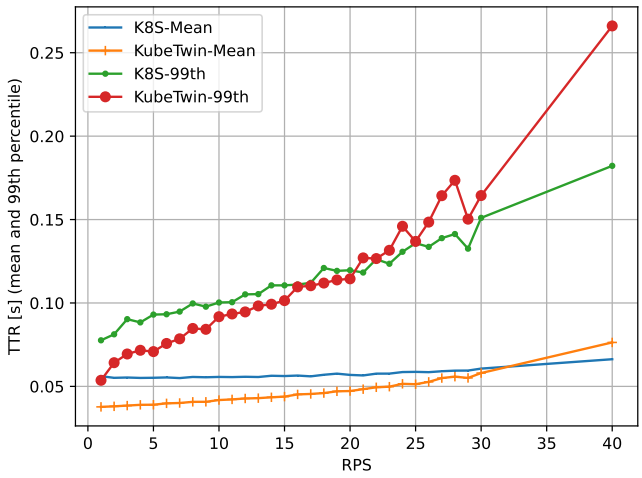


Figure 6.8: The distribution of the mean TTR value for 5000 requests under different RPS values using a 3 replicas deployment and the RPS 20 model.

These preliminary results demonstrate KT capacity to reenact the behavior of a microservice-based application running on a real K8s cluster. However, it is clear that to achieve a more accurate simulation of the system, it is necessary to improve the characterization of the response time of a single microservice, possibly leveraging a suitable ML technique. Specifically, the prediction of a continuous random variable (target) with conventional FFNs for regression often leverages the minimization of the Mean Squared-Error (MSE) cost function, providing an approximation of a target value with its conditional average, limiting its description. A more complete description of a random variable, in particular for targets with high variability, may be evaluated through the conditional Probability Density Function (PDF) of the target data, given an input vector (features). To characterize the conditional probability of the target, a model that combines a conventional FFN and a mixture model can be adopted[4]. This model is called MDN and provides a general framework to approximate conditional density functions of targets by modeling the probability parameters as a function of the features [248].

Given a feature vector $\bar{x}$ and a generic random variable T the set of parameters that characterize it would be $(\theta_1, ..., \theta_n)$, making possible to write its PDF as:

$$p(t|\theta_1(\bar{x}), ..., \theta_n(\bar{x})) \tag{6.4}$$

[4]L. Manca, D. Borsatti, F. Poltronieri, et al., "Characterization of microservice response time in kubernetes: A mixture density network approach," in 2023 19th International Conference on Network and Service Management (CNSM), 2023, pp. 1–9. doi: 10.23919/CNSM59352.2023.10327842.

where t is a possible value for T. By omitting the parameters in the notation, it can be referred to as $p(t|\bar{x})$ simply. The MDN aims to model the conditional probability density as a linear combination (*mixture model*) of kernel functions:

$$p(t|\bar{x}) = \sum_{c=1}^{C} \alpha_c(\bar{x})\phi_c(t|\bar{x}) \tag{6.5}$$

where $p(\bullet)$ is the PDF of the target variable, C is the number of components of the kernel in the mixture, $\alpha_c(\bar{x})$ is the mixing coefficient of the c-th kernel (it represents the prior probability of that kernel conditioned on $\bar{x}$), $\phi_c(t|\bar{x})$ is the c-th kernel (representing the conditional density function of the target t, conditioned on $\bar{x}$). The model allows different kernel choices (e.g., Normal, Uniform, Exponential, or other distributions); in the following, Weibull distribution was opted as the kernel of choice, which is characterized by two parameters, γ and k, called *scale* and *shape*, respectively. The Weibull distribution represents a proper choice since it generalizes the Exponential distribution and is suitable for describing non-negative random variables, such as the time between occurrences of events.

The generic Weibull kernel conditioned to $\bar{x}$ is:

$$\phi_c(t|\bar{x}) = \begin{cases} \frac{k_c(\bar{x})}{\gamma_c(\bar{x})} \left(\frac{t}{\gamma_c(\bar{x})}\right)^{k_c(\bar{x})-1} e^{-(t/\gamma_c(\bar{x}))^{k_c(\bar{x})}}, & t \geq 0 \\ 0, & t < 0 \end{cases} \tag{6.6}$$

Hence, in the mixture model, each term is characterized by a set of three parameters (mixture parameters): $\alpha_c(\bar{x})$, $\gamma_c(\bar{x})$ and $k_c(\bar{x})$, assumed to be unknown continuous functions of the features $\bar{x}$. In order to model the unknown functions:

- the first stage of the model is a conventional FFN, which takes as input a vector of features $\bar{x}$ and transforms them through proper weights $\bar{w}$, providing as output a vector $\bar{y}(\bar{x}; \bar{w})$;
- the output vector of the first stage is provided as input of a second stage represented by a mixture model.

In the mixture model, the transformed features $\bar{y}(\bar{x}; \bar{w})$ are processed using appropriate activation functions to model the mixture parameters (i.e., mixing coefficients, scales, and shapes). The final output of the model turns out to be the conditional probability density of the target (Equation (6.5)). The combination of the two stages is referred to as a MDN, whose basic structure is represented in Figure 6.9.

The adoption of this model requires the definition of some hyperparameters (i.e., fixed parameters defining the specific implementation of the neural network), such as the number of mixture components C, the kernel functions, and the number of hidden layers and units in the FFN.

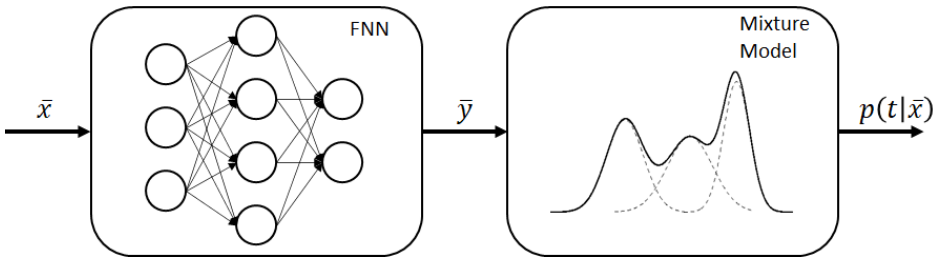


Figure 6.9: Structure of a MDN.

The mixing coefficients $\alpha_c(\bar{x})$ must satisfy the following condition:

$$\sum_{c=1}^{C} \alpha_c(\bar{x}) = 1 \tag{6.7}$$

which can be achieved by choosing the *softmax* activation function for these outputs, leading to the generic c-th mixing coefficient expressed as:

$$\alpha_c = \frac{e^{y_{\alpha_c}}}{\sum_{i=1}^{C} e^{y_{\alpha_i}}} \tag{6.8}$$

where y_{α_c} represents the FFN output related to the c-th mixing coefficient.

The scale $\gamma_c(\bar{x})$ and shape $k_c(\bar{x})$ parameters must satisfy the conditions:

$$\gamma_c(\bar{x}) > 0 \qquad k_c(\bar{x}) > 0 \tag{6.9}$$

They can be achieved by choosing the Exponential Linear Unit activation function:

$$\gamma_c = \mathrm{ELU}(y_{\gamma_c}) + 1 \tag{6.10}$$

$$k_c = \mathrm{ELU}(y_{k_c}) + 1 \tag{6.11}$$

where y_{γ_c} and y_{k_c} represent the FFN outputs related to the scale and shape of the c-th component, respectively. The Exponential Linear Unit is defined as:

$$\mathrm{ELU}(y) = \begin{cases} y, & y > 0 \\ e^y - 1, & y \leq 0 \end{cases} \tag{6.12}$$

The weights $\bar{w}$ in $\bar{y}(\bar{x}; \bar{w})$ are learned during the training of the MDN through a training set $\left\{\bar{x}^{(q)}, t^{(q)}\right\}$ of cardinality m by Maximum Likelihood Estimation. Its objective is to find the set of parameters for which the observed data (the training set) have the

highest joint probability. Assuming that the training examples are drawn independently from the PDF given by (6.4), the likelihood function of the set can be written as:

$$\mathcal{L} = \prod_{q=1}^{m} p(t^{(q)}, \bar{x}^{(q)}) = \prod_{q=1}^{m} p(t^{(q)}|\bar{x}^{(q)}) p(\bar{x}^{(q)}) \tag{6.13}$$

The maximum likelihood estimate is as follows:

$$\widehat{\bar{w}} = \arg\max_{\bar{w}} \mathcal{L}(\bar{w}) \tag{6.14}$$

From the likelihood, it is possible to derive the error function:

$$E = -\log \mathcal{L}(\bar{w}) = -\sum_{q=1}^{m} \log p(t^{(q)}|\bar{x}^{(q)}) \tag{6.15}$$

Formula (6.15) is called negative log-likelihood, in its expression the $p(\bar{x})$ factor is neglected since it does not depend on its parameters. Its minimization is equivalent to the maximization of likelihood. Taking into account the mixture model (6.5), the negative log-likelihood becomes:

$$E = \sum_{q=1}^{m} E^{(q)} = -\sum_{q=1}^{m} \log \sum_{c=1}^{C} \alpha_c(\bar{x}^{(q)}) \phi_c(t^{(q)}|\bar{x}^{(q)}) \tag{6.16}$$

where $E^{(q)} = -\log \sum_{c=1}^{C} \alpha_c(\bar{x}^{(q)}) \phi_c(t^{(q)}|\bar{x}^{(q)})$ is the error contribution of the q-th element in the training set.

Backpropagation is the standard procedure to minimize the error function. For this purpose, the gradient of the error function concerning the FFN output needs to be computed as reported in [249]. The software implementation of the backpropagation algorithm for this model can be inspired by regression FFN based on the MSE function. Modifying the error function is required to apply standard optimization procedures such as gradient descent. At the end of the training, the MDN can approximate the conditional density function of the target data given the input features, allowing it to have a probabilistic description of the data generation process. The MDN results are useful to derive specific moments (e.g., mean and variance).

Considering the representation of the microservice application, the cloud-native deployment scenario under analysis in the experimental work can be represented through a multi-layered queuing system. More generally, the implementation of the system can be described by several components.

- **Requests**: the system receives incoming requests sent by users, represented by a Poisson process with average request rate λ [req/s].

- **Replicas**: they represent multiple processing entities providing a given microservice (i.e., an independent portion of processes related to a more complete composite service) at a time; as they can operate in parallel and the execution is non-threaded, a section with n_{rep} replicas can contemporarily execute up to n_{rep} instances of microservices. The processing time related to a microservice performed by each replica can be represented by a random variable T_{ms} with unknown distribution.

- **Queues**: each replica in the system has its own queue in which requests are put on hold in case they cannot be immediately served.

- **Load balancer**: it routes the incoming requests to the available replicas; the simplest way to do it is to perform equal load balancing. In standard K8s clusters, the load balancing is probabilistic, with each replica having the same probability of being chosen to serve a request.

In a cloud-native deployment, a service is obtained through the chaining of several microservices, and this composition is strictly related to the kind of service provided to the users. With reference to a particular service, the structure described above can be stacked as many times as the number of microservices that compose the overall service, where each layer is responsible for a specific microservice. Figure 6.10 shows an example of a two-layered queuing system.

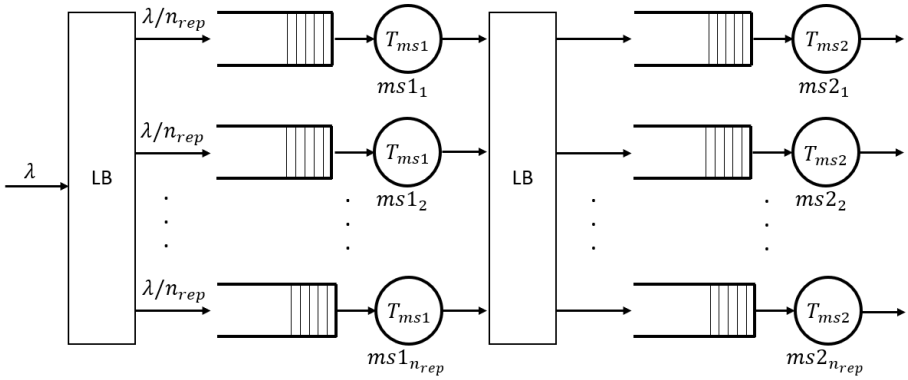


Figure 6.10: Example: a two-layered queuing system with n_{rep} replicas for both layers. In the first layer, on average, the load balancer (LB) equally distributes requests among replicas.

The multi-layered queue system during its operations (e.g., load balancing, queuing, microservices processing) introduces a random response time to the users, will be referred to as TTR. The distribution of the TTR is unknown and depends on the kind of service and the configuration of the system. In the experimental phase, a two-layered

queuing system with different values of $n_{\text{rep}} \in \{1, 2, 3, 4\}$ will be considered. The system is managed by the K8s platform, and replicas are deployed as containers distributed on a cluster. The designated service for that system is the previously defined image processing application, and the random processing times of a generic replica in the first and second layers are indicated as T_{ms1} and T_{ms2}, respectively.

The system employed for the measurements is a small two-node K8s cluster. Similarly to what previously presented, each node runs Ubuntu 20 LTS and is equipped with 4 vCPU, 8G RAM, and 70G of disk, with the only addition of the Calico plugin as Container Network Infrastructure, without any other custom module. Therefore, it could be considered as an ordinary standard deployment. Both microservices are Python-based software components that expose HTTP endpoints to enable interaction between them. Furthermore, both microservices log the duration of each HTTP request in their internal log. By retrieving these logs, it is possible to discover the time needed to complete each request, allowing us to measure the time a request spends in each of the two microservices. Then, by feeding these data to the MDN network, it is possible to obtain the statistical distribution of the two microservices response time. It is important to mention that this process could be highly automated with well-known K8s plugins, such as log-collecting tools like Fluentd or tracing operators like Jaeger (built-in in the Istio service mesh). Finally, a Python script to generated a fixed number of HTTP requests with a configurable Poisson arrival rate.

Thanks to these tools, several measurement campaigns of 1000 requests each were performed, at increasing arrival rates and with different application configurations. In detail, the number of replicas of each microservice increased from 1 to 4, with a varying behavior of each application component thanks to a "slow-down" factor sd. This value affects the response time of each microservice by forcing them to repeat their computing operations sd times before sending the reply. The reason behind this choice was that, in this way, it was possible to obtain microservices with different execution times in a controlled manner. Furthermore, by increasing these execution periods, it was possible to verify that some peculiar behaviors did not depend on the time scale of the measured execution time of a single function.

During experiments, sd has been scaled from 1 to 10, normalizing the request rate over the sd factor to keep the server utilization (i.e., the ratio of the arrival rate to the service rate) constant. Furthermore, the behavior of microservices was compared for multiple replica configurations with a sd factor equal to 10. As before, during the experiments the RPS value was tuned to keep the utilization constant (i.e., by multiplying the RPS value by the number of replicas considered). Focusing on the first microservice in the single replica configuration, Fig. 6.11 shows a decreasing mean for increasing values of the request rate, which applies for all the sd values. This behavior is counterintuitive, since it would be reasonable to believe that the average processing time of a microservice would either remain constant or increase with a higher number of incoming requests.

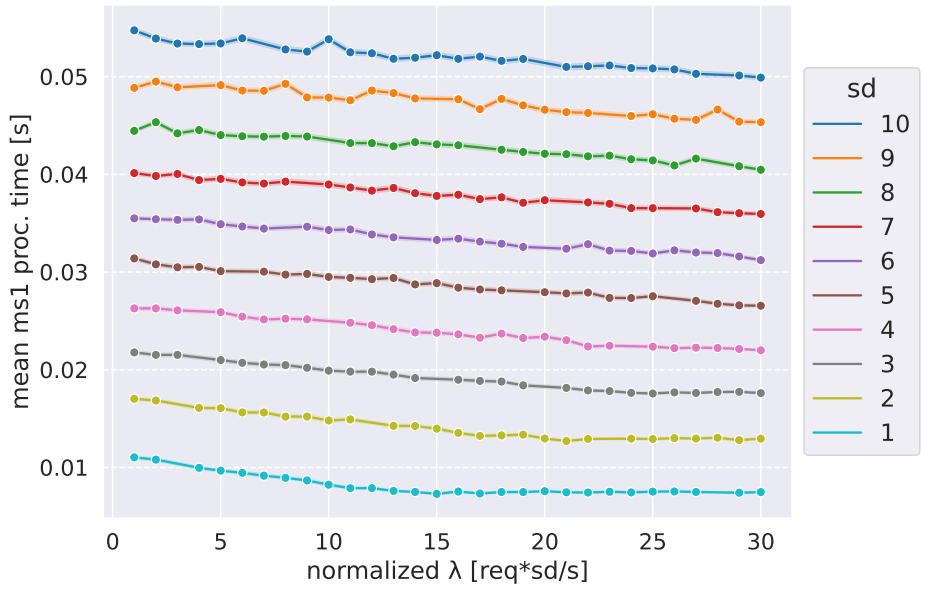


Figure 6.11: Comparison of MS1 mean proc. time, conditioned to RPS and *sd* factor from 1x to 10x (non threaded).

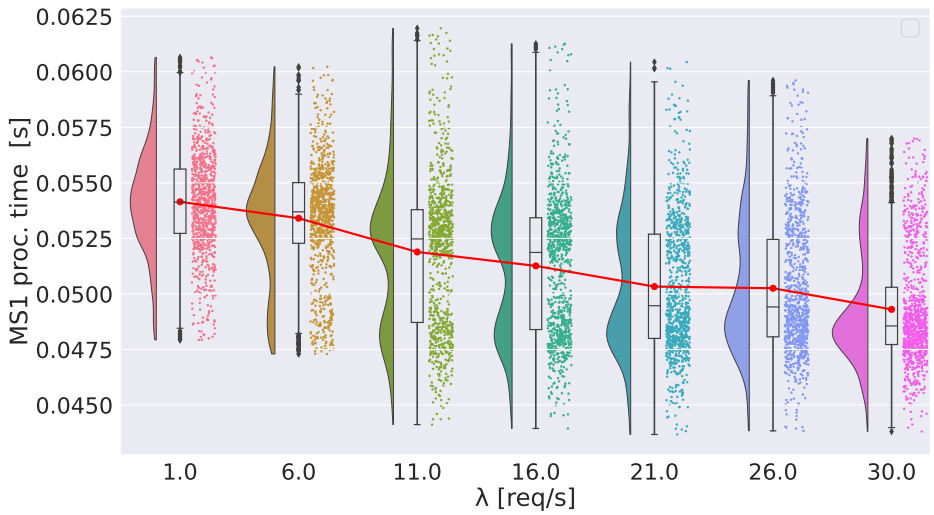


Figure 6.12: Raincloud plot [250] of MS1 proc. time, conditioned to RPS, for $sd = 10$ and one replica (non threaded).

As can be seen from the kernel density estimations depicted in Figure 6.12, the processing time of the first microservice exhibits strong multimodality in which the two main kernels vary their prior-probability with the arrival rate: for increasing arrival rate the prior-probability of the kernel with higher processing times decrease in favor of the

kernel with lower processing times. The reason behind these results requires further investigation, but it might depend on specific CPU allocation policies that try to optimize highly active processes. This result was one of the main reasons that motivated the search for modeling tools capable of learning these hidden dependencies. The first microservice shows the same dependency even in the case of multiple replicas (Figure 6.13). In particular, for the cases with one and three replicas, the difference in the mean processing time between the extreme values of request rate is about 8%. On the other hand, the same behavior is no longer appreciable for the second microservice, at least for the single replica deployment, as shown in Figure 6.14 where the processing times appear to have no relationship with the request rate. Going deeper into the statistics of the second microservice in the multi-replica deployment, Figure 6.15 shows an opposite behavior of MS2 with multiple replicas; this is highlighted in Figure 6.16 showing some processing times for the two replicas deployment. In general, Figures 6.13 and 6.15 show that for both microservices there are slight variations in response time by changing the number of replicas considered. Again, this was an additional element that pushed the research of a more complex modeling tool.

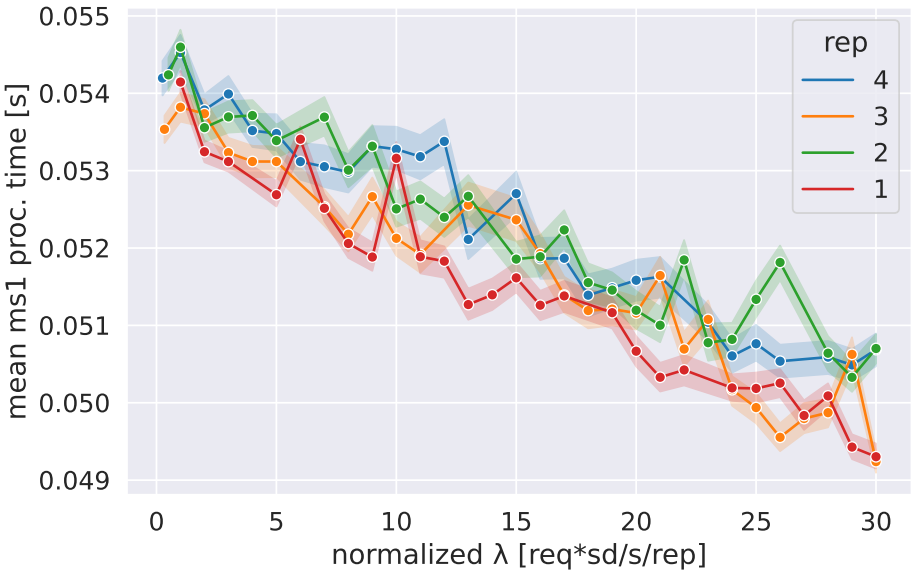


Figure 6.13: Comparison of MS1 mean proc. time (95% confidence interval) for $sd = 10x$. The processing time is plotted for different combinations of RPS values and numbers of replicas. The RPS (λ) is equally balanced among replicas.

As discussed previously, in this microservice framework the mathematical objective of the model is to approximate:

$$p(t_{ms,i}|\lambda, n_{\text{rep}}) \tag{6.17}$$

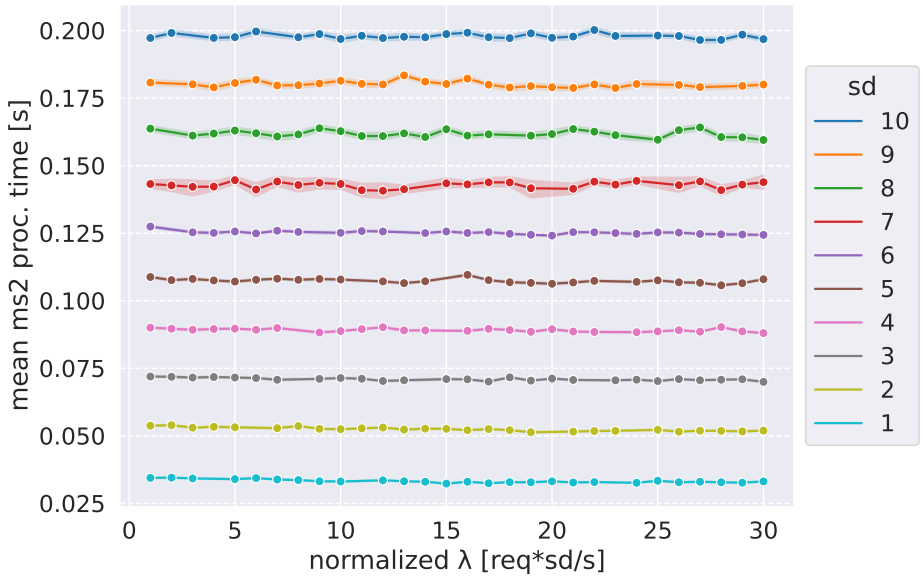


Figure 6.14: Comparison of MS2 mean proc. time, conditioned to RPS and *sd* factor from 1x to 10x (non threaded).

Where $p(\bullet)$ indicates the PDF, $t_{ms,i}[s]$ is the i-th microservice processing time ($i = 1, 2$), $(\lambda, n_{\text{rep}})$ are the request rate and the number of replicas available for that microservice, respectively. In other words, the model looks for and generalizes a probabilistic relationship (rather than a point estimate) between processing time, request rate, and number of available replicas, which proves useful for simulations and analysis by the DT.

The model is a FFN implemented with the Keras API [251] and the TensorFlow probability library [252], with the following characteristics:

- 2 input neurons: one input neuron for RPS and one for n_{rep};
- 4 mixture components: the approximated PDF is the superposition of 4 Weibull's PDFs;[5]
- 12 output neurons: the outputs represent the parameters of the components in the mixture model, considering that every Weibull's PDF is characterized by 3 parameters (the mixture parameter in the superposition, the scale and the shape);
- 2 hidden layers: each hidden layer has 8 neurons.

[5]During initial trials, the Normal, Gamma and Weibull PDFs were tried, then the latter was chosen because it provides the best fit among them.

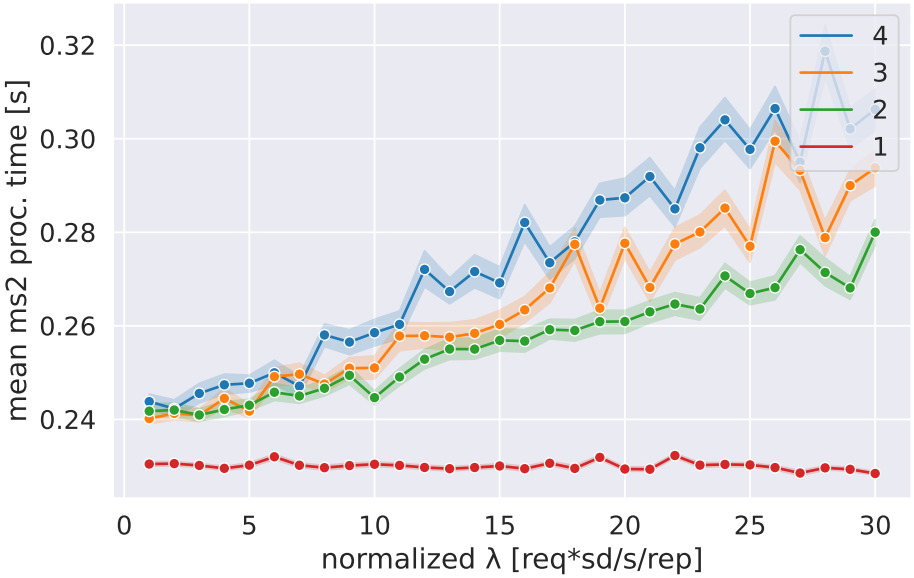


Figure 6.15: Comparison of MS2 mean proc. time (95% confidence interval) for $sd = 10x$. The processing time is plotted for different combinations of RPS values and numbers of replicas. The RPS (λ) is equally balanced among replicas.

The hidden neurons are set with *LeakyReLU* (Leaky Rectified Linear Unit) activation function. The activation functions of the output neurons depend on the parameter they represent, as previously discussed. At the end of the training phase, the microservice processing time is characterized in the following way:

$$p(t_{ms,i}|\lambda, n_{\text{rep}}) = \\ = \sum_{c=1}^{4} \alpha_c(\lambda, n_{\text{rep}}) \phi_c\left(t_{ms,i}|\lambda, n_{\text{rep}}\right) \tag{6.18}$$

The model architecture described was applied to both microservices MS1 and MS2 with the data set presented. The available data set was partitioned as follows: 70% for the training set, 20% for the validation set and 10% for test set. The performance of the model must be evaluated through a goodness of fit measurement, quantifying the disagreement between the prediction made by the model and a sample of observed values. The goodness of fit of the MDN model was evaluated using the Kolmogorov-Smirnov test [243] and the Wasserstein distance [253] in the test set, represented by 12 samples with 1000 elements each. More specifically, the Kolmogorov-Smirnov test compares the empirical Cumulative Distribution Function $F_n(t)$ provided by a sample with n elements with a reference Cumulative Distribution Function $F(t)$ through the Kolmogorov-Smirnov distance, defined as:

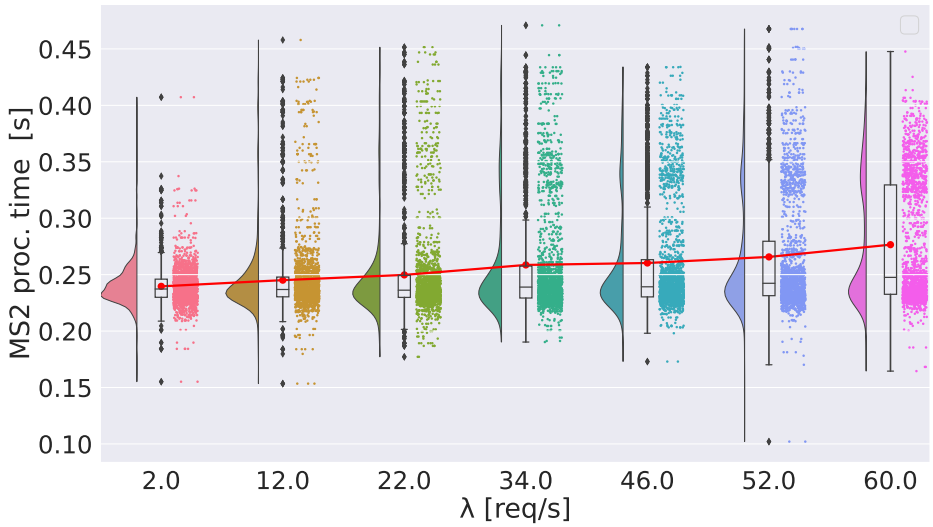


Figure 6.16: Raincloud plot of MS2 proc. time, conditioned to RPS, for $sd = 10x$ and two replicas (non threaded).

$$D_n = \sup_t |F_n(t) - F(t)| \tag{6.19}$$

The hypotheses of the test are:

$$H_0 : F(t) = F_n(t), \ \forall t \tag{6.20}$$

$$H_1 : F(t) \neq F_n(t), \ \text{for some } t \tag{6.21}$$

Under the null hypothesis, the Kolmogorov-Smirnov distance (Equation 6.19) is distributed according to the Kolmogorov distribution and converges to 0 in case $n \longrightarrow \infty$. The statistical significance of the test is represented by the p-value p, that is the probability of having a Kolmogorov-Smirnov distance at least as extreme as the one observed under the null hypothesis. By setting a significance level α (typically $\alpha = 0.05$), the null hypothesis H_0 is rejected in favor of the alternative H_1 in case $p \leq \alpha$, otherwise there is no evidence to reject H_0.

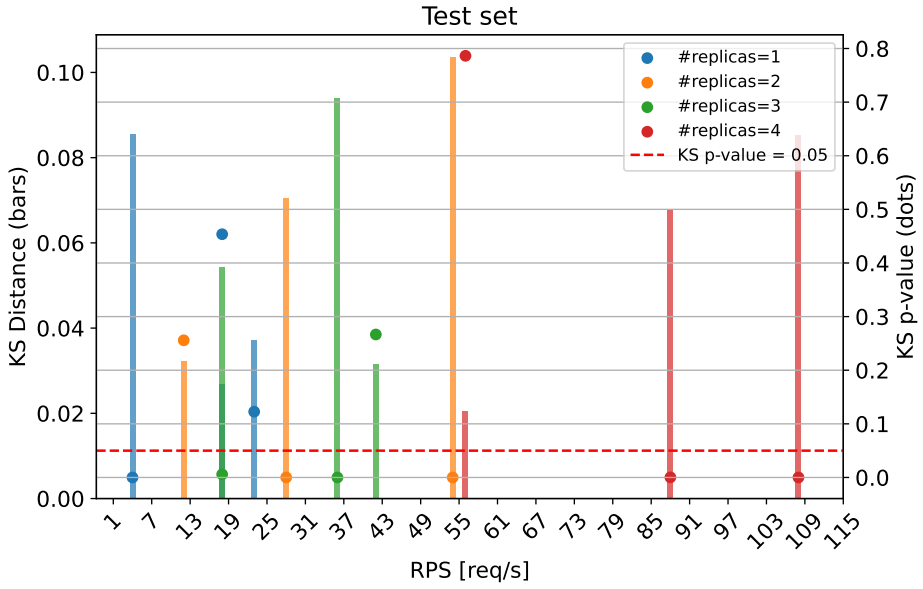


Figure 6.17: Performance evaluation of MDN model for MS1 in terms of Kolmogorov-Smirnov on the test set.

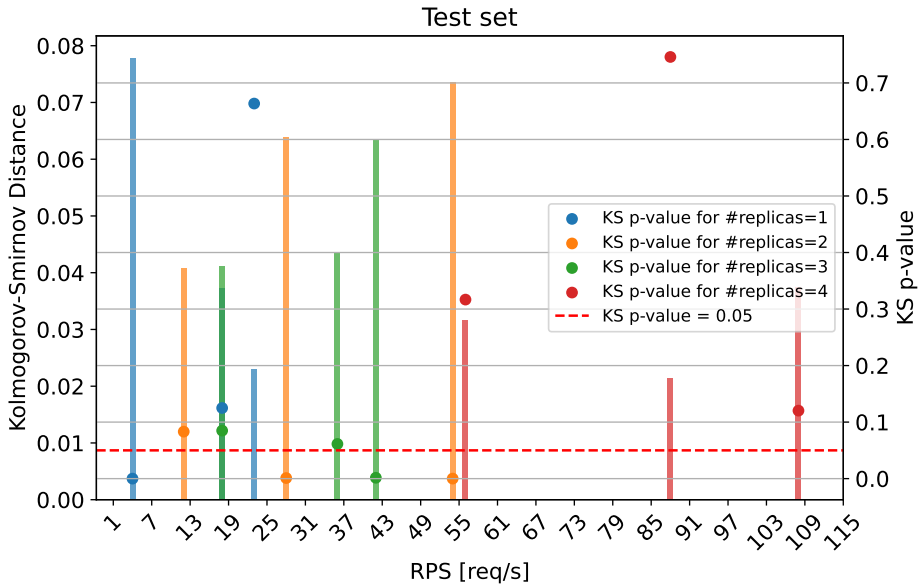


Figure 6.18: Performance evaluation of MDN model for MS2 in terms of Kolmogorov-Smirnov on the test set.

Figures 6.17 and 6.18 represent the performance of the MS1 and MS2 models on the test set, in terms of the Kolmogorov-Smirnov test. For MS1, despite most distances

being below 0.1, 7 out of 12 sets show that the null hypothesis is rejected with p-values below 0.05. For MS2 the maximum distance is below 0.08 and 4 out of 12 sets show that the null hypothesis is rejected. Since the Kolmogorov-Smirnov test compares the maximum distance between distributions, while the objective of our models is to generate samples to emulate microservices inside the DT, we introduced the Wasserstein metric to take into account the overall distance between test samples and samples generated by our models. In particular, this metric represents the minimum "cost" to transform the distribution from one sample to the other.

In the one dimensional case, for two empirical measures P and Q with respective samples $t_1, \ldots, t_n$ and $u_1, \ldots, u_n$ of cardinality n (in ascending order), the p-Wasserstein distance is defined as:

$$W_p(P, Q) = \left(\frac{1}{n} \sum_{i=1}^{n} \|t_i - u_i\|^p \right)^{1/p} \tag{6.22}$$

In our framework, the 1-Wasserstein distance implemented in [254] was employed to quantify the distance between a sample in the test set for a couple $(\lambda, n_{\text{rep}})$ and the sample generated by the model with the same couple as input. In order to relate this metric with the test sample under analysis, the Standardized Wasserstein distance has been introduced as the ratio between the 1-Wasserstein distance previously described and the standard deviation of the test sample under analysis. In this way, the metric can be interpreted as the "distance in terms of standard deviations".

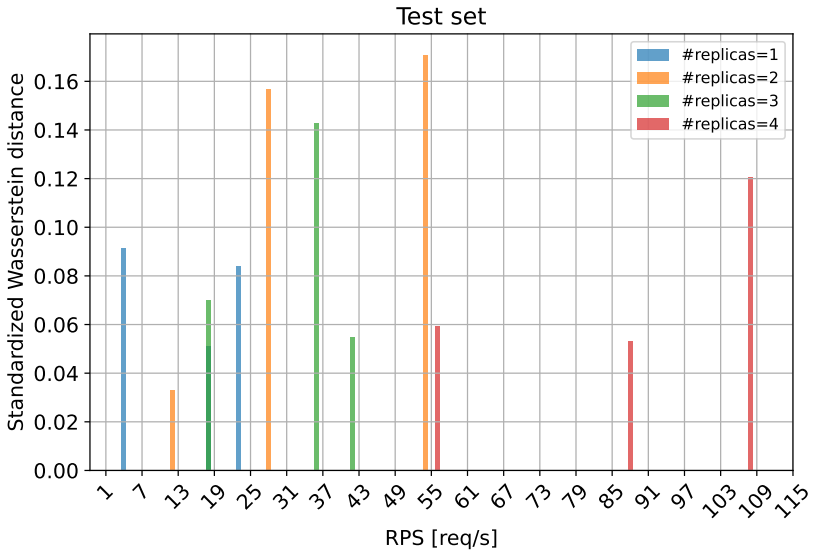


Figure 6.19: Performance evaluation of MDN model for MS1 in terms of Standardized Wasserstein metric on the test set.

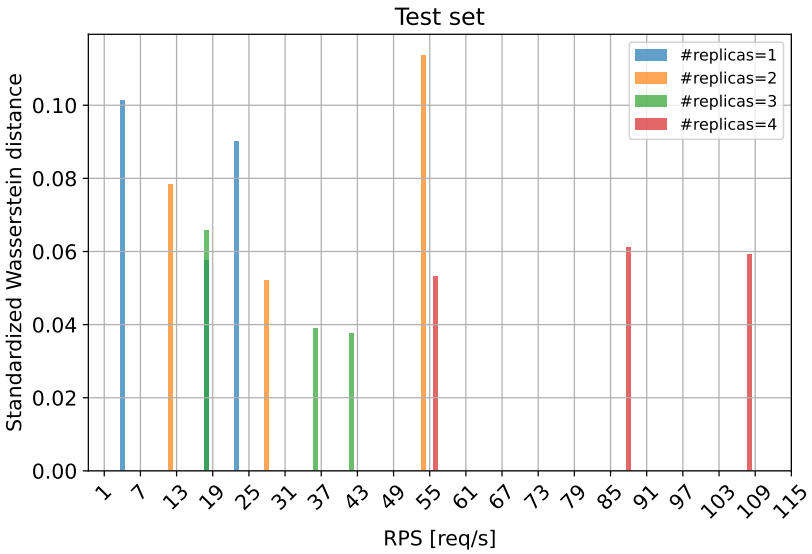


Figure 6.20: Performance evaluation of MDN model for MS2 in terms of Standardized Wasserstein metric on the test set.

Figures 6.19 and 6.20 show performance evaluation performed with Standardized Wasserstein distance. For MS1 all the distances are below 0.2, with 8 out of 12 test samples showing a distance below 0.1. The performance in MS2 shows even smaller distances, with 10 out of 12 test samples below 0.1. Since most of the test samples show distances close to or below 0.1, the expectation related to the error in the approximation of the true microservices samples with the ones generated by our mixture models in the DT are expected to be relatively small compared to the variability of the true data. However, in a production scenario, the approximation error restrictions need to be related to the specific application requirements.

After obtaining the model of the two microservices from the MDN, they have been plugged into the KT simulator to compare the overall application response times from the K8s cluster with the ones generated by the simulator. In this experiment, KT reenacted the same request generation patterns as during the measurement campaigns. At each request generation time, the simulator samples from each microservice distribution a response time using as input the RPS of the last eight requests (tunable parameter). Then, the simulator computes the total request time by summing up the two distribution samples and the queueing time of the request. The details on how KT manages the internal queues are described previously and can be found in [173].

Figure 6.21 depicts the mean and 99th percentile of the total application response time measured from the K8s system and KT, with $sd = 10$ and a single replica for each microservice. The results show that the simulator can reenact the system behavior with good accuracy. In detail, the average MSE measured between the two is around

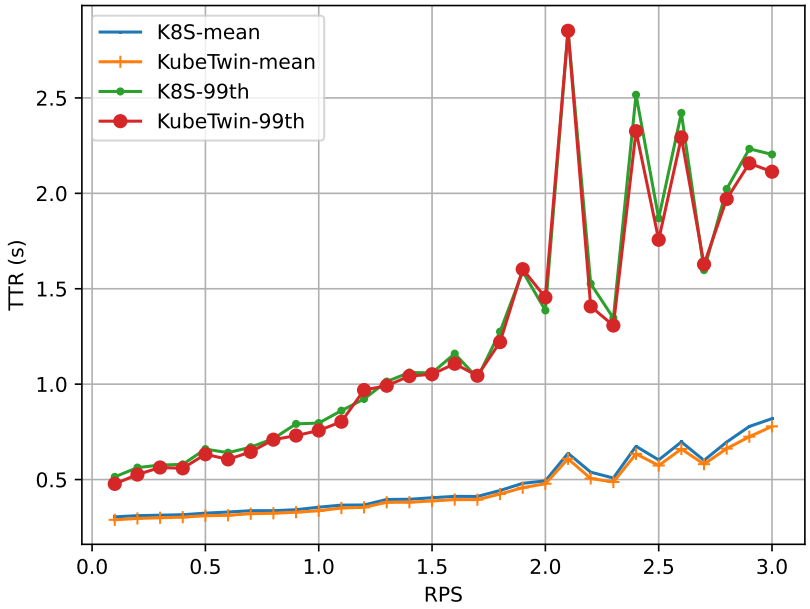


Figure 6.21: Comparison between the mean and 99th percentile response time of the real application and KT, with $sd = 10$ and 1 replica for each microservice.

0.00263 for the mean response time, as reported in Table 6.2. We repeated the test with the same configurations but considering the microservices modeled with the methods presented previously (i.e., a three-component Gaussian Mixture Model fitted over the data measured with RPS = 1). Applying this modeling method, we obtained a much larger MSE value of about 0.20881 for the mean response time. These results show a strong improvement in the performance of KT, thus proving the soundness of the proposed characterization of response time with a MDN approach.

Finally, the same tests were replicated with three replicas. Figure 6.22 reports the results of the MDN-based model with this application configuration. Compared with those obtained with a single replica, there is a slight performance degradation. This could be caused by the randomness introduced by the load-balancing feature of K8s. Since vanilla K8s distributes the request among available replicas using equiprobable iptables rules, some microservices might receive higher request rates than others, thus leading to various response times. However, the comparison with the method presented previously still shows a substantial MSE reduction.

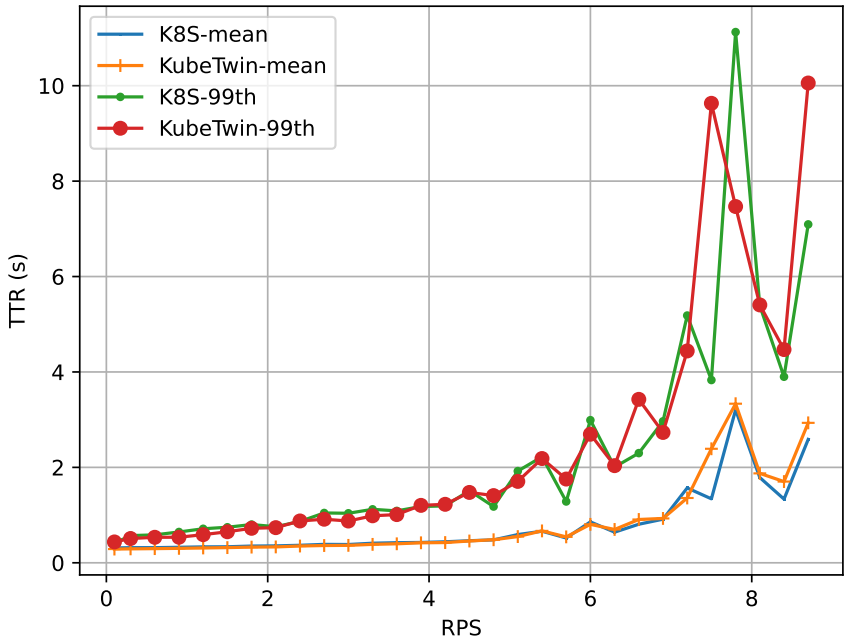


Figure 6.22: Comparison between the mean and 99th percentile response time of the real application and KT, with $sd = 10$ and 3 replicas for each microservice.

Table 6.2: Average MSE comparison between the proposed solution and the Gaussian Mixture presented in [173] for both mean and 99th percentile of the TTRs.

	MDN		Gaussian Mixture[173]	
n_{rep}	Mean	99th percentile	Mean	99th percentile
1	**0.00263**	**0.00423**	0.20881	1.02729
3	**0.04830**	**1.95080**	10.38479	56.5381

6.2.5 KubeTwin Framework Evaluation

To evaluate KT, a set of experiments was devised on the image recognition application used to model the microservice response time discussed previously. The main objective of these experiments is to validate the KT simulation framework as a valuable solution to experiment with different K8s deployments. To this end, a performance comparison between several simulations used different deployment configurations to find a suitable deployment for the use-case case described above. Then, a second experiment validated the horizontal scaling reenactment within the KT framework.

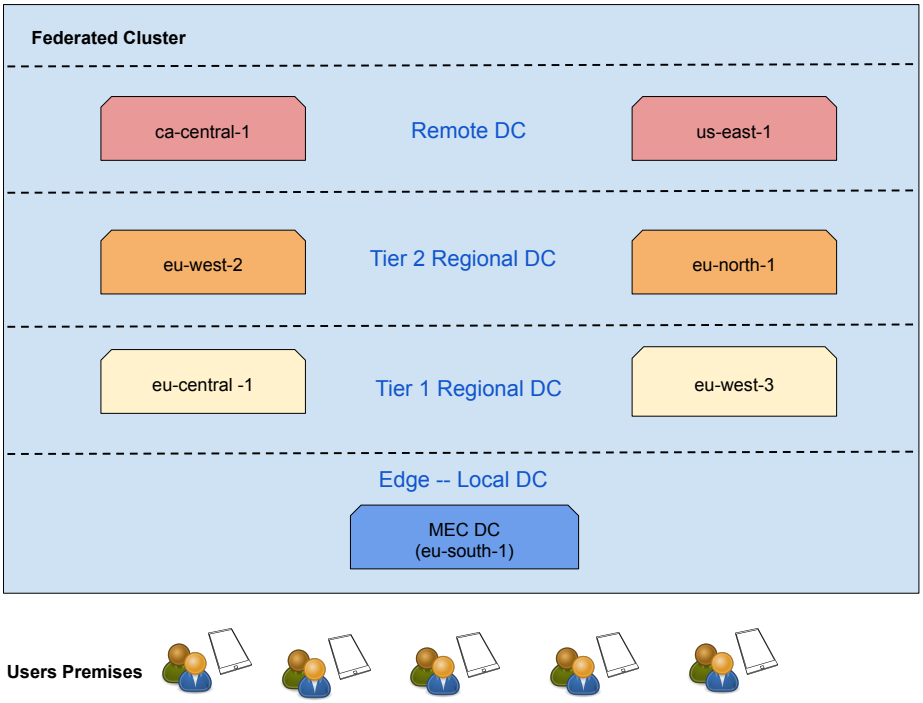


Figure 6.23: The federated cluster scenario described for the experiments. Users are located in the proximity of the MEC DC, thus can benefit from a reduced communication latency. Other tiers provide computing nodes to distribute the application load.

The image recognition service is available to users located in the smart city of Milan, in which a MEC DC is available to its citizen. Fig. 6.23 illustrates the MEC deployment in a federated cluster scenario considered for the experimental evaluation. Specifically, a small-size MEC DC in Milan, modeled as a single Local DC in the proximity of the user premises, provides computing capabilities at a reduced communication latency, two Tier 1 Regional DCs, and two Tier 2 Regional DCs. Furthermore, the models adopted for cloud clusters can be beneficial to scale the application performance when needed, e.g., when the resources at lower tiers are saturated, by providing enhanced and on-demand computing capabilities at the price of higher latency. These resources were identified as the data centers flagged in the Remote DC tier. The assumption that users are located in the proximity of the Local DC is a typical example of a small edge data center located at the smart city borders. There, users play with some "augmented reality application", which continuously calls the image recognition service to identify objects within the camera frames collected using the application. Communication latency between the different locations is modeled according to the CloudPing based model described previously. With regard to computing capabilities at the different tiers, there is availability of 25 nodes in the MEC DC, 100 nodes in Tier 1, 150 nodes in Tier 2,

and 200 nodes in the Remote DC tier. For the following experiments, we assume that the computing nodes at Local DC, Tier 1, and Tier 2 have the same amount of CPU and memory resources, which correspond to a constant value of 100 for both CPU and memory. This value refers to a standard amount of resources required by a machine with medium computational power (i.e., an i7 core). Instead, we assume that the computing nodes in the Remote DC are twice as powerful, i.e., they have a value of 200 for both CPU and memory. This metric allows to have a good granularity in modeling the computational capacity gap between the different layers of the cluster and to evaluate the behavior of K8s applications in a CC scenario, in which the availability of more powerful computing nodes is typically offset by increased communication latency. Finally, at the scheduling level, higher scheduling priorities (node affinity) are assigned to the computing nodes that reside at the lower tiers. Specifically, nodes at the Local DC have the maximum scheduling priority, then each tier at a higher level presents a decreasing scheduling priority. This choice defines a MEC-first filter-and-score procedure that will make the KTScheduler first selects computing nodes close to the user premises. This scheduling priority is different from the default, which would try to equally distribute the replicas among all available nodes in the cluster, regardless of the performance in terms of latency.

For the first validation,KT can be used to identify a suitable configuration for the scenario of interest. Specifically, with this analysis, there is the need to identify the number of replicas associated with each microservice. This procedure can be used to find a suitable deployment configuration that could satisfy the requirements of a relatively static scenario, which is characterized by a constant workload of user requests. Furthermore, KT can demonstrate its effectiveness as a DT, whose purpose is to help service providers identify a suitable configuration for their real-world system. For the following experiments, user activity was modeled as an aggregated flow of requests with constant intensity. More specifically, the interarrival times were defined by sampling from a random variable with exponential distribution to simulate a workload of 100 RPS. Moreover, we set the request generator to send a total of 10,000 requests, considering 10 seconds for the simulation warmup and 10 seconds for the simulation cooldown. Each microservice has a dedicated KTReplicaSet, which maintains a stable set of replica pods running at any given time. The main objective is to evaluate different deployment configurations by analyzing the TTR. To this end, multiple simulations run with different numbers of replicas (from 1 to 20) and TTR of each request was collected to calculate the mean and the 99th percentile TTR. For the sake of clarity, these values (i.e., mean and 99th percentile TTR) include the time spent by requests while traveling to the simulated locations (i.e., the time spent transmitting the request between users and data centers). With this experiment, we could visually identify those deployments capable of fulfilling a given Service Level Indicator. In this case, a target 99-th percentile TTR was set to 60 ms and the objective was to look for a deployment configuration capable of satisfying the 60 ms Service Level Indicator.

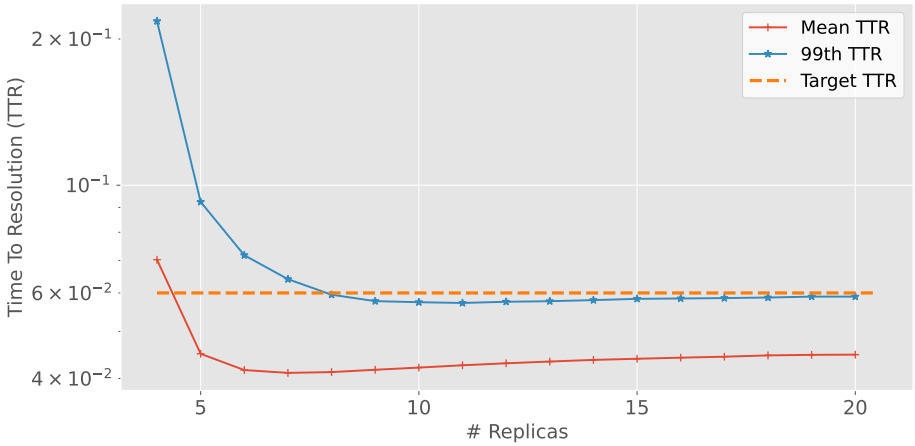


Figure 6.24: Mean and 99th percentile TTR at different numbers of replicas

The results of this validation are visible in Figure 6.24, which shows the TTR and 99th percentile TTR while varying the number of replicas on the x-axis. the results for the configurations with less than four replicas are excluded from Figure 6.24, which were out of scale. Specifically, looking at Figure 6.24, it is easy to note that configurations with up to 8 replicas struggle to meet the application requirements (60 ms 99th percentile TTR) even if the mean TTR is below the threshold, since the 99th percentile TTR of the requests is still above the given target. In contrast, configurations with 9 and more replicas show a decreasing trend in both the mean and the 99th TTR. Furthermore, Figure 6.24 also shows that over scaling the number of replicas does not improve the performance in terms of the mean and 99th percentile TTR. Therefore, it is easy to visually determine that a deployment configuration with 9 replicas for each microservice (MS1 and MS2) can fulfill the stringent 60 ms Service Level Indicator. Additional replicas could be instantiated to deal with temporary fluctuations in the application workload. For this reason, a deployment configuration with 10 replicas has been selected for both MS1 and MS2. As another interesting result, considering the small number of pods to allocate, the edge computing resources at the Local DC are sufficient to deploy up to 20 replicas for each microservice. Therefore, KT can effectively reenact also custom K8s scheduling policies, such as the one specified that tries to instantiate all pods in the MEC DC first. This would allow service providers to estimate the deployment costs for multiple configurations, e.g., shifting from the different tiers of the CC depending on the prices of the computing resources, but also on the specific requirements of these applications.

In a second validation, the objective is to verify if KT can support HPA. Starting from the previous experiment, a more workload-aggressive scenario has been defined. More specifically, for this experiment, the aggregated flow of user requests was mod-

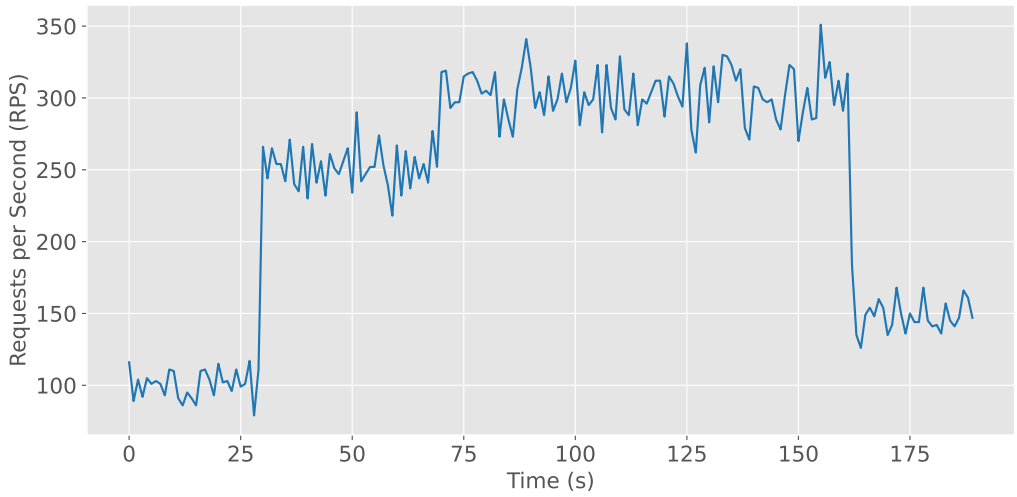


Figure 6.25: The distribution of the requests during simulation time.

eled as the summation of several flows of constant interarrival times to stress the performance of the current deployment. This experiment should validate whether KT can increase the number of replicas to meet the increased workload. It is worth noting that this kind of experiment, which stresses the application by applying an increasing workload, can find many employments. For example, a service provider may need to test its K8s deployment to configure the lower and upper bound for the K8s HPA.

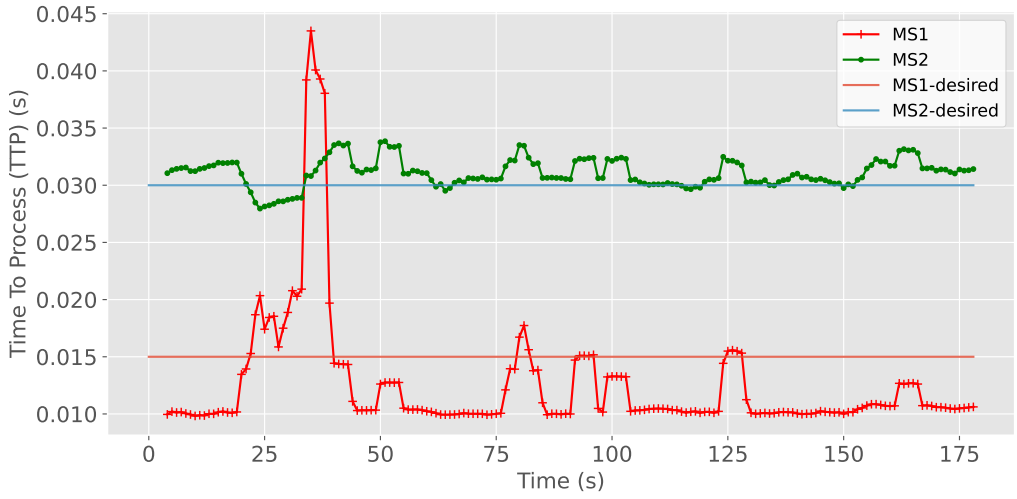


Figure 6.26: Average Time-to-Process requests per micro-service during the simulation time compared to the desired Time-to-Process.

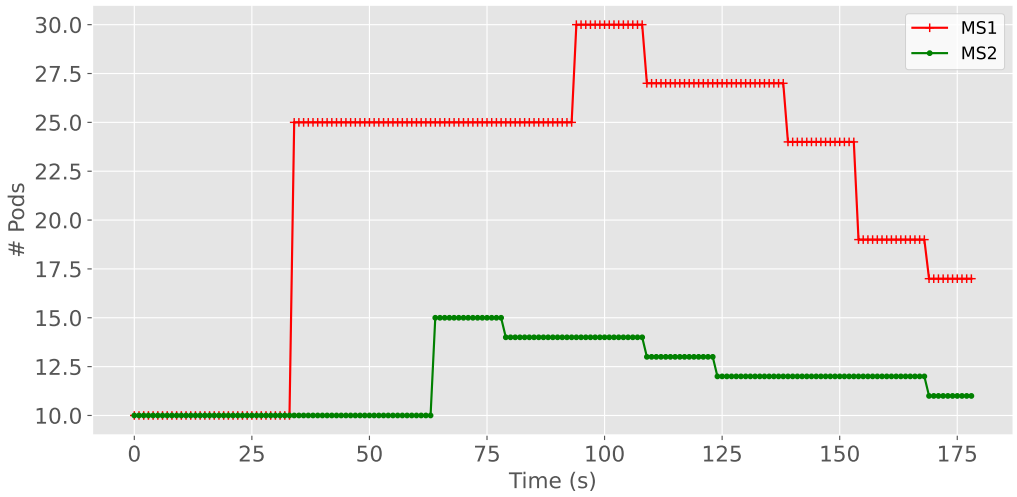


Figure 6.27: Distribution of the number of pods for each microservice during the simulation time.

For this scenario, KT reenacted the processing of 130,000 requests. Specifically, a baseline workload consisted of 100,000 requests at 100 RPS, which was then increased after 30 seconds since the beginning of the simulation by an additional workload of 20,000 requests at 150 RPS. Finally, the simulation considered an additional 10,000 requests at 50 RPS starting 70 seconds after the beginning of the simulation. For clarity, Figure 6.25 illustrates the workload distribution, which shows a spike in requests starting from the 25th second of the simulation and ending around the 160th second of the simulation, when the workload of 20,000 requests at 150 RPS terminates. This workload pattern defines an unexpected spike in the number of requests that would undermine the performance of the current deployment configuration. To reenact autoscaling features, a KTPodScaler component was specified in the KT configuration for each microservice that would periodically check the desired performance scale of our targets. In addition, each KTPodScaler considered the parameters for $minReplicas = 10$ (the deployment configuration selected at the previous step) and $maxReplica = 50$, and a period of 15 seconds (the default interval according to K8s documentation). From the most basic perspective, the KTPodScaler analyzes the ratio between the desired metric value and the current metric value (see Eq. 6.1), skipping any scaling action if the ratio is within the $[0.90, 1.10]$ interval. Figures 6.26 and 6.27 report the results of the validations, which show respectively the average Time To Process along with the desired one and the number of instantiated pods measured during the simulation time. Specifically, the average Time To Process indicates the average time for processing a request, calculated as the difference between the finished processing time and the time at which the request arrived at the container, whereas the desired Time To Process for each microservice is defined as its expected processing time (i.e., when the system is not overloaded)

plus a 20% tolerance, which is the maximum value that we are willing to accept. Figure 6.26 shows that the most affected microservice is MS1, which is the entry point for the application workflow. Specifically, there is a notable increase in the Time To Process, which is mainly due to a non-optimal number of instantiated replicas and a very strict desired metric, illustrated with straight orange (MS1) and blue (MS2) lines in the figure. On the other hand, MS2 is less affected by the increased workload, as it starts processing each request only after MS1 finishes the initial processing. Therefore, the Time To Process trend for MS2 is almost linear with some increase during the simulation time. To improve the performance of the current configuration, the KTPodScalers intervene to reduce the Time To Process for MS2 and MS1 by allocating more replicas, as visible in Figure 6.27. Specifically, this happens around the 30th second of the simulation, i.e., in correspondence with the increased workload, where the Time To Process starts to increase for all microservices. This behavior happens because the current deployment (10 pods for MS1 and 10 pods for MS2) cannot process the increased amount of incoming requests, i.e., these requests get queued. However, as soon as the KTPodScalers intervene to activate new replicas, the Time To Process for the illustrated microservices starts to decrease, stabilizing for the rest of the simulation. The activity of the KTPodScalers is also visible in Figure 6.27, which shows that the number of associated pods changes to cope with the increased request workload and to respond to performance degradation. The number of instantiated pods for MS1 goes up to 30, and the one for MS2 goes up to 15 to address the notable increase in the Time To Process, while it gradually decreases when the workload goes back to 150 RPS as expected. Additionally, while the average Time To Process illustrated in Figure 6.26 is the result of all requests processed during simulation time, the KTPodScalers take the scaling decisions by considering the metric calculated over 15 second intervals, as described previously.

Finally, the distribution of the instantiated replicas for the most intensive deployment was retrieved from the seconds 105 to 120 of the simulation time and analyzed. During this time window, the KTPodScaler instantiated 30 pods for MS1 and 14 pods for MS2 among the Local DC and the Regional DC at Tier 1. Specifically, this distribution is the result of the scheduling policy that tries to exploit computing resources at the lowest tiers first. It should be noted that this scheduling policy would define deployment as more performing in terms of TTR because of the reduced communication latency given by the proximity of these computing resources to end-users.

6.3 Chapter Summary

This chapter presented KT, a comprehensive framework designed to implement DTs of K8s-based deployments. KT emulates the K8s orchestration functions and networking behaviors to be used as a guideline for service providers to accurately evaluate the impact of complex and large-scale K8s deployment scenarios. It allows the definition of

applications through a declarative description, which is semantically equivalent to K8s, to reenact and assess them through both generic and application-specific (and user-defined) Key Performance Indicators. KT enables its adopters to leverage the DT to identify optimized deployment configurations and evaluate different scheduling policies in complex computing scenarios where resources can be distributed at many levels, such as edge servers and cloud data centers.

KT has been evaluated in a reference use case reenacting a distributed MECapplication deployment. The proximity of these applications to the network's access part offers reduced latency and increased bandwidth. Furthermore, MEC offers standardized APIs that applications can consume to get real-time radio access network information. The application deployment we considered includes a hybrid computing scenario with a three-tier MEC environment and two remote Cloud data centers, representing a relevant use case to test the capabilities of KT. This evaluation has shown that KT can replicate complex and large-scale IT infrastructures at both the application and orchestration platform levels. This allows for capturing Key Performance Indicators that are essential to gaining an understanding of how the system operates.

To make this evaluation reliable, a statistical description method for response time based on MDN has been investigated, which combines the use of conventional FFN with characterization by means of mixture models. experimental results show how MDN allows to accurately estimate the statistical distribution of the response time of an illustrative microservice-based application. Section 6.2.5 showed how a MDN-based model for the estimation of the statistical distribution of the response time of microservices could improve the performance of simulation-based approaches like KT, improving its effectiveness as a DT for K8s environments.

The soundness of the KT framework as a DT for K8s deployments is thus demonstrated, and it presents interesting research opportunities for future investigation, as highlighted in its applications described in the next chapters.

Chapter 7

Application of Kubernetes Digital Twins for Chaos Engineering

CE is the discipline of injecting computing and network faults, such as increased network latency and unavailability of computing nodes, into an IT system to help developers identify problems that could arise in a production environment and tackle them. Several tools have emerged to ease the application of chaos engineering to complex IT systems, leveraging microservice and container-based applications deployed on Kubernetes. However, applying such tools requires several phases to be put into practice, from defining a steady state to establishing an effective response plan if something goes wrong. At the same time, because of how dynamic and complex K8s deployments are, unexpected faults can emerge over time, thus making it more challenging to find a reliable configuration for a K8s application. Therefore, there is a need for more efficient solutions to test the resilience of an application and to make teams more aware of their ecosystem dynamics [255]. If the adoption of CE can help to solve this task, there is still a need to put in practice several phases, from defining a steady state to establishing an effective response plan if something goes wrong. Consequently, the application of effective fault countermeasures and predicting the exact behavior of an application before its actual deployment in a real environment can involve very costly operations, paving the way for the application of DTs. In fact, DT methodologies can bring many advantages to a wide range of analysis [256], [257] and ease the application of techniques such as CE [232]. Specifically, by creating a virtual replica of a running K8s application, it is possible to run what-if scenario investigations that evaluate both the impact of injected faults and also the effects of mitigation policies.

This chapter focuses on investigating the combination of CE and DT methodologies to improve the resilience of container-based applications deployed atop K8s. Specifically, it presents TELKA, a RL-based scheduler for K8s. TELKA can retrieve metrics and monitoring information and intervene in case of performance degradation due to computing and network faults. TELKA adopts KT to learn how to mitigate adverse

events of faults, such as node failure or increased communication latency. KT enables to run multiple what-if scenario analyses during which the application is affected by injected faults. To deal with these faults, TELKA interacts with KT to learn a policy that can improve the resilience of the entire application. In this way, instead of interacting with the physical K8s application, TELKA learns by interacting with a DT, thus reducing the learning time and the operation costs related to the application of CE.

7.1 The TELKA Scheduler

CE principles have been demonstrated to be very effective in increasing the resilience of IT systems. Concerning the specific case of K8s, it is interesting to evaluate the impact of injected faults and to evaluate best practices for mitigating these effects. In this regard, TELKA combines DT, CE, and RL methodologies to define a smart scheduler for K8s that can mitigate the effects of computing and network faults[1]. Specifically, TELKA learns how to intervene in case of faults by interacting with a virtual replica of a K8s application defined with KT, a framework we devised to define virtual replicas of K8s applications and reenact their behavior in a computing scenario of interest presented and described previously in this thesis. This DT approach enables running a what-if scenario analysis in less time than running the same analysis on the physical testbed. In addition, it is easier to define complex computing scenarios and simulate injected faults. Therefore, TELKA embraces this approach to run multiple what-if scenario analyses and learn the best mitigation policies using RL algorithms. During each what-if scenario analysis, TELKA interacts with the DT of the K8s application as a scheduler that reallocates K8s pods running on a node affected by a CE fault. Specifically, by simulating the internal control loop of K8s, KT can identify when a computing node stops responding, using the default probing mechanisms implemented in K8s. When this occurs, TELKA can intervene to suggest K8s where to reallocate the evicted pods.

The whole reallocating approach is depicted in Fig. 7.1, showing how TELKA interacts with the DT. During each simulation run, KT injects a collection of faults. Then, when an error has been identified on the node, KT replicates the kubelet's action to proactively terminate the pods and reclaim the resources on that node. After each pod has assumed the evicted state, the node where the pods were running is deallocated (i.e., excluded from the cluster), and its resources are retrieved. At this point, instead of replacing the evicted pods, TELKA analyzes the status of the available nodes and then decides where to assign the evicted pod. Regarding the modeling of the RL environment, a state includes the evicted pod to reallocate (see Fig. 7.1) and a list of the

[1] M. Zaccarini, D. Borsatti, W. Cerroni, et al., "Telka: Twin-enhanced learning for kubernetes applications," in 2024 IEEE Symposium on Computers and Communications (ISCC), 2024, pp. 1–6. doi: 10.1109/ISCC61673.2024.10733736

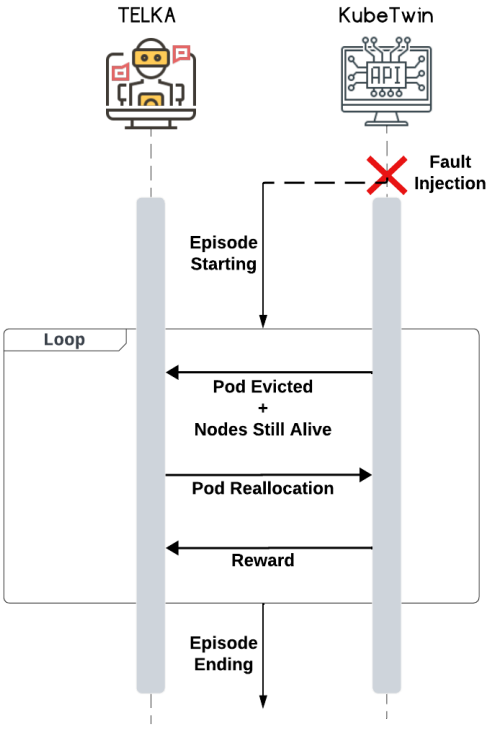


Figure 7.1: The workflow between TELKA andKT. TELKA tries to reallocate evicted pods due to injected faults.

remaining nodes available, where each node includes its identifier, the amount of CPU and memory available, and the cluster identifier, i.e., the cluster where the node resides. On the other hand, the action to be performed at each step is the selection of the node's identifier where to allocate the evicted pod. Once TELKA selects the node, it sends this information to KT, which applies the action and returns a reward to TELKA. This process is repeated until the simulation (an episode) ends.

Concerning the reward, the following definition has been adopted:

$$R = \begin{cases} 1, & \text{if the pod is reallocated successfully.} \\ -0.2, & \text{if the target node does not have enough resources.} \\ -0.5, & \text{if the target node does not exist or it is unreachable.} \end{cases}$$

where 1 corresponds to a successful reallocation operation, -0.2 to an unsuccessful action that reallocates the pod to a node where there are not enough computing resources, and -0.5 to those actions that select a non-existing or unreachable node. Moreover, the

agents earn an additional reward at the end of the episode. Unlike other rewards, which focus only on orchestration operations, the final reward assigns an additional "bonus" based on overall performance. Specifically, it is designed to improve actions that can lead to a higher percentage of satisfied requests and a reduced average TTR. In this way, the agent should learn an orchestration policy capable to deliver a higher QoS to the application's user. When the agent receives the last reward, the socket is closed, and the episode ends. Finally, concerning the implementation details, the communications between KT and TELKA was modeled via a UNIX socket created in each episode when the simulation started. Specifically, when KT detects a node failure, it establishes a connection with TELKA, which then sends the data of each evicted pod, one at a time, together with the information of all nodes that are still running in the clusters. This communication model makes TELKA relatively flexible and efficient, and it was implemented to allow a preliminary evaluation of this solution. Finally, the architecture behind this mechanism enables it to potentially be adapted to a RESTful HTTP model similar to the one employed by the K8s API.

To evaluate TELKA, a CC scenario composed of multiple clusters located in the edge, fog, and remote cloud layers was considered. Specifically, as represented by Figure 7.2, the scenario defined a total of seven different clusters of computing resources, i.e., groups of computing nodes: one at the edge (close to the end users of the application), four fog computing clusters, and two clusters in remote cloud locations, which can be exploited to scale the application when needed. Concerning computation availability, each cluster at the edge and fog layer has 10 available nodes. Instead, each cluster located in cloud data centers has 20 nodes, with a double resource availability compared to nodes at the edge and fog layers. To ensure the right fairness and fidelity in reenacting the physical K8s application, the image recognition application previously described and modeled with MDN through KT was considered as a reference and ran on top of the computing resources. The workload was composed of an aggregated flow of requests with a constant intensity, with an interarrival time defined by sampling from a random variable with exponential distribution, enabling a simulation of the processing of nearly 80000 requests. The communication latency between cluster locations was shaped with randomly generated measures that consider latency data through connection tests conducted among several AWS DCs available via CloudPing.

Table 7.1: Fault Injection Model used in TELKA Experimental Evaluation

Fault Type	Clusters Involved	Fault Description	Fault Start Time
Added delay	C1 and C2	Added 150 ms of communication delay between clusters from edge cluster C1 to a fog cluster C2	start_time + 25 seconds
Added delay	C2 and C4	Added 100 ms of communication delay between two fog clusters C2 and C4	start_time + 50 seconds
Added delay	C1 and C6	Added 75 ms of communication delay between clusters from edge cluster C1 to a cloud cluster C6	start_time + 30 seconds
Node Injection	C1	2 node injections on cluster C1	start_time + 20 seconds
Node Injection	C3	4 node injections on cluster C3	start_time + 20 seconds
Node Injection	C5	5 node injections on cluster C5	start_time + 20 seconds

Then, to apply CE techniques, Table 7.1 defines the faults considered, which KT injects during each simulation run. Specifically, we envision two different types of faults: the first consists in increasing the communication latency between pairs of clusters, and the second type injects faults to reenact node failures. With these fault configurations, the use case achieves two different objectives: i) simulate the unavailability of computing nodes and thus triggering the rescheduling of TELKA and ii) modify the scenario's condition, i.e., there is a performance difference in terms of communication latency between reallocating a pod in one cluster or a different one. During the training phase, the faulty nodes in each cluster involved in the experiment are randomly selected. This enables the generation of a considerable number of different scenarios, taking advantage of the potential of the DT paradigm as a training ground for RL techniques.

This evaluation aims to verify if TELKA can learn an efficient policy that reallocates evicted pods after the fault injection described in Table 7.1. Moreover, to ensure a complete and fair analysis, TELKA's results have been compared with two other approaches: a random approach, which simply selects a random node from the ones available and a greedy one that, at each step, always chooses the node with the largest available resource pool. As part of the TELKA's evaluation, it is interesting to consider a

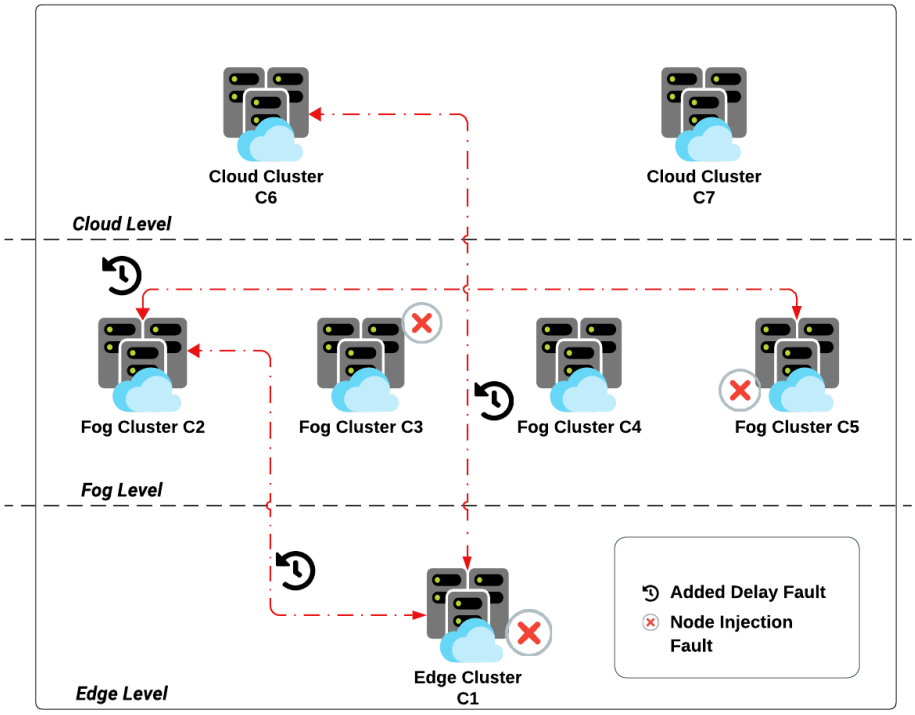


Figure 7.2: TELKA evaluation scenario and fault injections summary.

performance comparison with two well-known RL algorithms, PPO and DQN, to decide which could perform better for this task. For both PPO and DQN, we leverage the standard versions of the algorithms provided by Stable Baselines3 have been considered, according to experiments presented throughout this thesis. The reason behind this choice is associated with their simplicity and effectiveness demonstrated in various scenarios, such as managing resources in the CC.

As for the training phase, the process included a total of 50.000 steps, where each step represents a single pod reallocation by the agent. Figure 7.3 shows the evolution of the average episodic return obtained by both DQN and PPO agents during training, i.e., the reward earned at the end of each episode. It is easy to note the increasing PPO trend, which demonstrates its good learning capabilities and the reward structure's effectiveness in tackling the problem. On the other hand, DQN seems to remain stuck between 10 and 12, suggesting an inferior capability to handle the given scenario. The superior performance of PPO is also visible in Figure 7.4, which shows the ratio of successful pod reallocation during training. PPO can consistently reallocate more than 85% of the evicted pods, while DQN stays frozen around 80% throughout the training process.

As preliminary validation, models were tested on the training scenario and com-

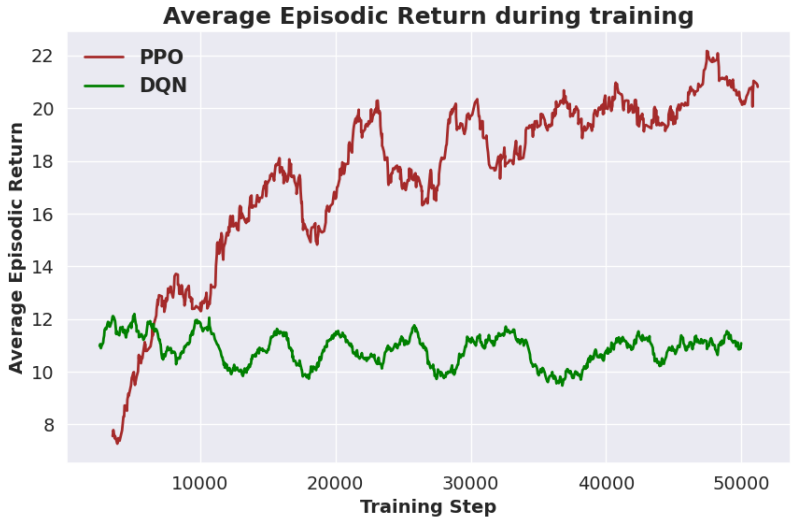


Figure 7.3: Average DQN and PPO Episodic Return in 50000 steps of training.

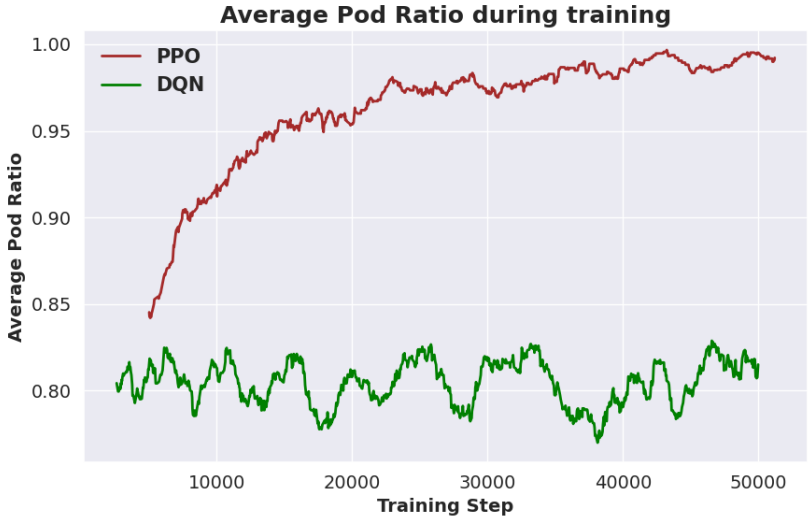


Figure 7.4: Average DQN and PPO Pod Ratio in 50000 steps of training.

pared DQN and PPO with the random and greedy approaches. Specifically, a total of 25 test episodes were performed in the baseline scenario, assessing the average TTR of each individual request and the ratio of pods successfully reallocated by the agent to the total pods evicted due to simulated fault injections. For clarity, TTR refers to the time required to satisfy an HTTP request to the image recognition application. Table 7.2 outlines the mean value, the standard deviation, and the 90th percentile for both

Table 7.2: Results obtained by different approaches in 25 testing episodes.

Approach	Metric	Mean Value	Standard Deviation	90th Percentile
PPO	TTR [s]	**0.2413**	**0.0304**	**0.2785**
PPO	Pod Ratio	0.9934	**0.0195**	**1.0**
DQN	TTR [s]	0.2686	0.0630	0.3184
DQN	Pod Ratio	0.3967	0.1272	0.5747
Random	TTR [s]	0.2543	0.0389	0.3152
Random	Pod Ratio	0.8107	0.0892	0.9147
Greedy	TTR [s]	0.2642	0.0472	0.3247
Greedy	Pod Ratio	**0.9937**	0.0196	**1.0**

the TTR (measured in seconds) and the pod reallocation ratio collected across the 25 testing episodes for each approach under analysis.
Considering the TTR result, PPO consistently outperforms the others, being the only one under 0.25 seconds. This is a notable result, especially if considered along with the pod reallocation ratio outcome. Indeed, the PPO agent achieves an average ratio of successfully reallocated evicted pods of 99.34% (100% taking into account the 90th percentile) during all testing runs. Contrarily, DQN seems to be the worst approach in this scenario. Despite a competitive TTR, it fails to achieve a reallocation ratio higher than 40%, thus indicating that, in this specific use case, an on-policy approach like PPO appears to be more suitable than an off-policy approach. Finally, random and greedy approaches demonstrate predictable behaviors in their testing trials. Given the abundant resources across all combined clusters, the random strategy does a good job of reallocating the evicted pods. However, it does not reach the performance of the greedy strategy, which, very similarly to PPO, maintains a ratio very close to the optimal.

A second more challenging experimental evaluation was designed to assess the strength and adaptability of the approaches. Specifically, the number of nodes subject to fault injections increased, with a total of 6 node failures at the Edge level on Cluster 1, 4 node failures at the Fog level on Clusters 4 and 5, and 10 node disruptions at the Cloud level on Cluster 6. Consequently, the approaches under analysis had to deal with an increased number of faults almost three times larger than the baseline scenario, proving their capabilities to deal with unseen and more challenging conditions. Analogously to Table 7.2, Table 7.3 presents the results obtained by all approaches in terms of mean value and standard deviation over 25 testing episodes.

Even in this experiment, PPO outperformed the alternatives, obtaining the highest pod reallocation ratio and the lowest TTR, demonstrating a very good adaptability

Table 7.3: Results obtained by different approaches in 25 testing episodes in heavy fault scenario.

Approach	Metric	Mean Value	Standard Deviation	90th Percentile
PPO	TTR [s]	**0.2804**	**0.1129**	**0.3376**
PPO	Pod Ratio	**0.9095**	0.0826	**1.0**
DQN	TTR [s]	1.8098	2.4012	4.8841
DQN	Pod Ratio	0.2933	0.0646	0.3727
Random	TTR [s]	0.3325	0.1396	0.4994
Random	Pod Ratio	0.8005	**0.0537**	0.8713
Greedy	TTR [s]	0.4698	0.6149	0.8827
Greedy	Pod Ratio	0.8836	0.0919	0.9892

to the mutations introduced. On the other hand, DQN continued to struggle significantly, failing to effectively reallocate more than 30% of the evicted pods and exhibiting a TTR nearing 2 seconds. Meanwhile, the random technique is notably impacted by the increased fault injections and manifests a notable drop in the ratio (almost 60%). The greedy heuristic, while slightly better in maintaining a high reallocation ratio compared to PPO, saw a noticeable increase in its TTR, almost doubling. This deterioration is likely due to the more aggressive fault model of this scenario, which forces a significant reliance on the remaining active cloud nodes, i.e., the ones with higher resources available. Therefore, this could be conceivably the central cause of the TTR increase, given that these nodes are the farthest from users and consequently require more time to process each request and send the response back.

7.1.1 Deep Sets Exploitation in TELKA

While TELKA showed promising results in reallocating evicted pods, its preliminary implementations suffered from scalability issues, as the RL agent could only operate effectively in scenarios with the same number of nodes seen during training. To overcome this limitation, DS can be incorporated in TELKA, generalizing its operation on different numbers of nodes[2].
Specifically, the details of the data that compose the set used in this work are indicated in Table 7.4. By interacting with KT through a UNIX socket created at each episode,

[2]M. Zaccarini, F. Poltronieri, D. Borsatti, et al., "Chaos engineering based kubernetes pod rescheduling through deep sets and reinforcement learning," in NOMS 2025-2025 IEEE Network Operations and Management Symposium, 2025, pp. 1–7. doi: 10.1109/NOMS57970.2025.11073590.

Table 7.4: The structure of the features used as an input of DS.

Set	Description
Pod Information	1: ID number of the pod. 2: Original node where the pod was allocated. 3: Resources required to run.
Node Information	1: ID number of the node. 2: CPU available (in MilliCPU). 3: Memory available (in RAM). 4: Operational status.

the agent collects information about both the evicted pods (including their identifiers, the original node where it was running, and resource demands) and the remaining operational nodes (their identifiers, current status, and free resources in milliCPU and RAM). To ensure a fair comparison with the original setup of PPO and DQN, the reward definition is the same as adopted in the previous implementation [258]: successful pod reallocations receive a positive reward, a minor penalty is applied if the agent picks a node with not sufficient resources, and a larger penalty is imposed if the node is unreachable or unavailable. The reason behind this last case is that, even though TELKA receives a list of available nodes at each step, it is still possible that a selected node becomes unavailable. This is because the probing mechanism that checks the status of the available node does not refresh after every step. This is considered reasonable, as it is consistent with the behavior of K8s, which performs these checks periodically (by default every 15 seconds).

To validate the capabilities of the DS-RL architectures, the previous cluster distribution in the CC was considered. More details of the characteristics of each cluster in the scenario are indicated in Table 7.5. The ones that belong to the first two layers of the CC (Edge and Fog layers) have half the total computing nodes and resources available for each one of them compared to the clusters placed at the Cloud level. However, relying too heavily on the resources at the Cloud level would result in a degradation of the performance, especially if metrics like TTR, i.e. the time required by the K8s application to satisfy an HTTP request, are considered.

To ensure a fair comparison, the experimental evaluation was also analogous to the one conducted for the first implementation of TELKA. As a first step, both PPO-DS and DQN-DS were validated in the same scenario used for their training process and compared their results across a total of 25 testing episodes. Secondly, a much more demanding optimization problem was designed to exploit the generalization capabilities that characterize DS. Specifically, an increased number of faults were injected into the system, causing a variation in the size of the input manipulated by the algorithms.
Table 7.6 illustrates the workload adopted in both experimental evaluation scenarios,

Table 7.5: The multi-cluster configurations described for the TELKA use case, which depicts the configuration for the nodes available in clusters.

Name	Tier	Node No.	Node CPU Resources	Node Mem. Resources
eu-south-1	Edge	10	100	100
eu-central-1	Fog	10	100	100
eu-west-3	Fog	10	100	100
eu-west-2	Fog	10	100	100
eu-north-1	Fog	10	100	100
ca-central-1	Cloud	20	200	200
us-east-1	Cloud	20	200	200

Table 7.6: The configuration for the statistical distributions of the workload, in number of requests per second, sent to the application during the experiments.

Start Time	Exponential λ	Number of requests
Start	50	50,000
Start + 30 seconds	25	20,000
Start + 70 seconds	35	10,100

which generates approximately 80.000 requests in a total of 1000 seconds of simulation, including 10 seconds of initial warm-up and a corresponding cool-down period. The workflow is composed of three different flows, each starting within the first 100 seconds of simulation at a constant intensity. The interarrival time follows an exponential distribution with unique rates assigned to each flow. Starting from the previously described use case, both the PPO-DS and DQN-DS versions were evaluated under different conditions. Specifically, these experiments aim to validate the ability of both approaches to learn a policy that reallocates pending and unscheduled pods as a consequence of fault injections. Leveraging DT and DS, this approach results in savings in computational resources and facilitates the management of the system in the event of sudden and potentially disruptive faults, offering a more proactive response than K8s mechanisms like autoscaling. The training phase conducted was 15.000 steps long, where each step corresponds to a pod reallocation attempt by the TELKA agent. Specifically, Figures 7.5, 7.6, and 7.7, respectively, show the episodic return (i.e. the reward earned at the end of each episode), the percentage of successfully reallocated pods, and the average TTR

earned by both RL solutions. Both DQN-DS and PPO-DS effectively handle the problem from different perspectives while benefiting from the proposed reward structure, converging to a high accumulated reward, achieving 90-100% reallocation of evicted pods, and reducing the average TTR during training.

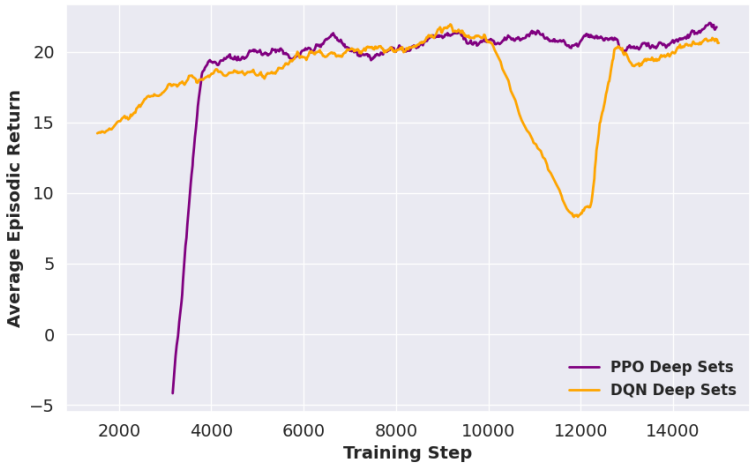


Figure 7.5: Average episodic return during training.

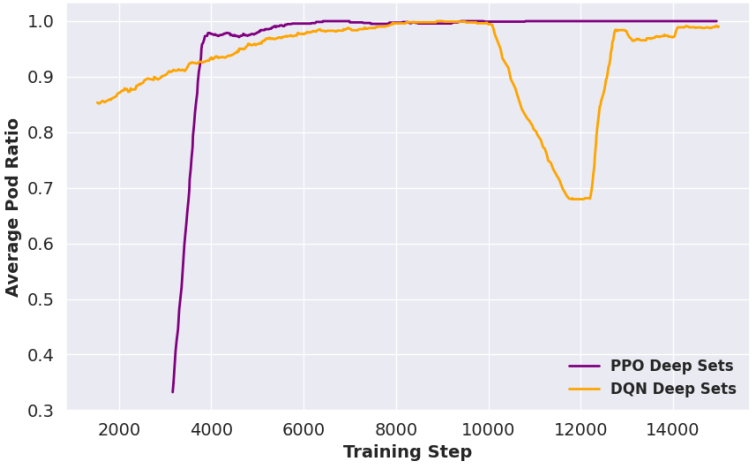


Figure 7.6: Average pod ratio during training.

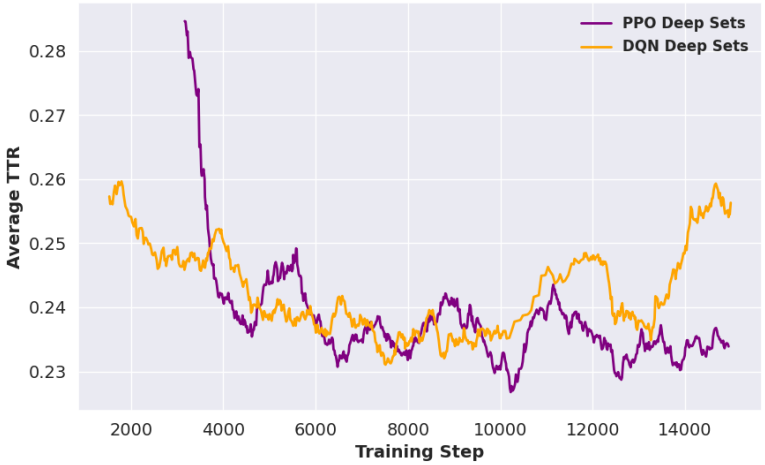


Figure 7.7: Average TTR during training.

Figure 7.8: Performance metrics during training for PPO-DS and DQN-DS.

Compared to the previous work, the training length was reduced to 15.000 steps instead of potentially risking early convergence, which may limit the search process. Despite this, DQN-DS experienced a temporary deterioration of the episodic return and pod reallocation ratio around 12.000 steps, suggesting the detection of a new search space region that could potentially lead to a more profitable solution. Conversely, PPO-DS showed limited improvements after the first 5000 steps, making 15.000 steps a reasonable choice overall. Looking more closely at the metrics, DQN-DS shows remarkable and consistent performance improvement compared to its predecessor, increasing by approximately 60% its episodic return (from 12 to 20) over 15.000 steps, and its pod reallocation ratio from 0.8 to nearly 1.0. Regarding PPO-DS, while it shows a similar trend to its earlier version under the same conditions, it achieves these results with significantly fewer steps and reduced computational effort.

As indicated in Table 7.7, the mean value, the standard deviation, and the 90th percentile have been collected for each one of the considered metrics: TTR and pod reallocation ratio. In the baseline scenario, both approaches obtained very good results considering all the metrics included in the analysis. Specifically, DQN-DS achieved an almost identical pod reallocation compared to PPO-DS, which successfully reallocated the totality of evicted pods during testing. At the same time, DQN-DS was able to make decisions that resulted in slightly better performance in terms of TTR, as visible in the numerical results illustrated in the table. Thus, with the additional reward associated with the performance of the system as defined in the previous work, DQN-DS achieved a higher overall reward. Compared with the results shown in Section 7.1, both DS versions outperform the previous methods, with DQN-DS achieving a no-

Baseline Scenario			
Approach	**Metric**	**DS Impl. Mean**	**DS Impl. 90th perc.**
PPO	TTR[s]	0.2278 (0.0142)	**0.2395**
PPO	Pod Ratio	**1.0 (0.0)**	**1.0**
DQN	TTR[s]	**0.2179 (0.0458)**	0.2607
DQN	Pod Ratio	0.9833 (0.0372)	**1.0**

Table 7.7: Comparative results of DS and previous approach in the baseline scenario

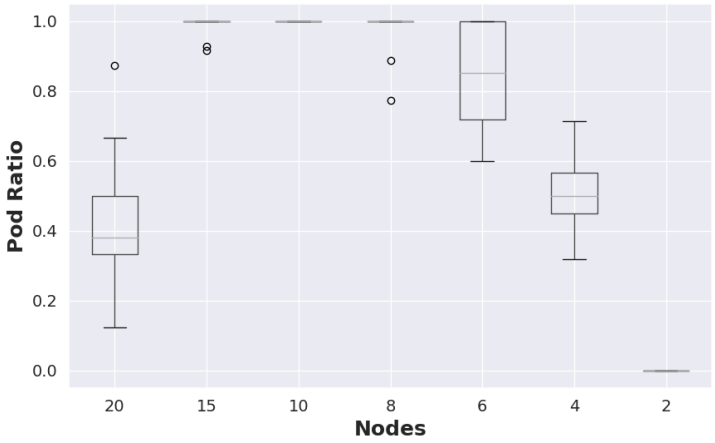
Heavy Fault Scenario			
Approach	**Metric**	**DS Impl. Mean**	**DS Impl. 90th perc.**
PPO	TTR[s]	**0.2372 (0.0096)**	**0.2452**
PPO	Pod Ratio	**1.0 (0.0)**	**1.0**
DQN	TTR[s]	0.2406 (0.0276)	0.2633
DQN	Pod Ratio	**1.0 (0.0)**	**1.0**

Table 7.8: Comparative results of DS and previous approach in the heavy fault scenario

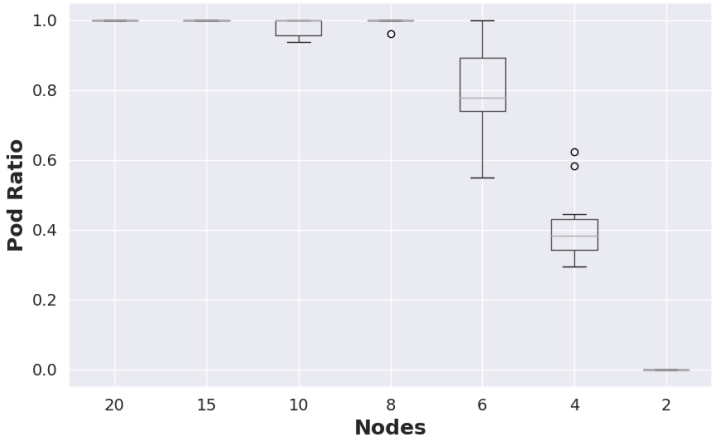
table 60% improvement in pod reallocation. These findings highlight the potential of DS compared to standard versions of PPO and DQN in such scenarios.

Despite the alterations introduced, the results associated with the heavy fault scenario, illustrated in Table 7.8, once again confirm the validity of the proposed approach. Both algorithms maintain very good performance in all metrics evaluated. The increase in the TTR can be attributed to the large number of faults and consequently fewer available nodes. Between them, those allocated at the lower level of the CC provide the main performance contribution from a TTR perspective. Therefore, even though both algorithms achieve near-perfect pod reallocation ratios, the TTR and the total reward show a slight decrease compared to the baseline scenario. Similarly to the experiments in the baseline scenario, Table 7.8 highlights the analogies and discrepancies between the two versions of each approach. On the first hand, regarding PPO-DS, there is an improvement of 10% in the pod reallocation ratio and nearly 15% if considered TTR values. On the other hand, once again DQN-DS is the approach that introduces the most significant improvements. The version in this work can reach a TTR which is nearly 87% better than the DQN standard implementation and 70% in the pod reallocation metric.

This final experiment assess how well DQN-DS and PPO-DS handle clusters with



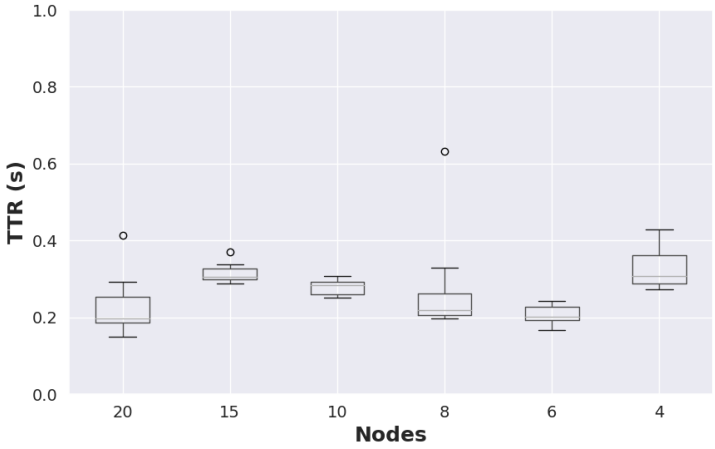
(a) PPO-DS.



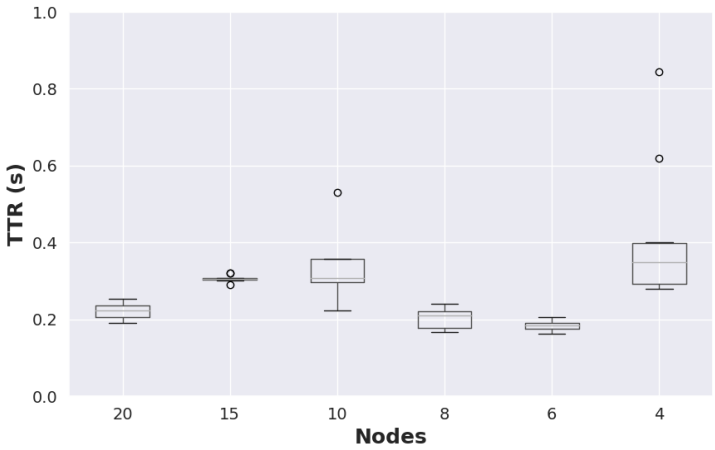
(b) DQN-DS.

Figure 7.9: Statistical distribution of the Pod Reallocation Ratio across 10 tests by varying the number of nodes per cluster from 20 to 2.

varying number of nodes. After injecting the fault, both approaches restore the service, thereby estimating the fewest resources needed to overcome the fault model. To do so, baseline fault scenario was leveraged to run experiments with multiple cluster sizes, identifying a viable number of nodes for system recovery.
Figures 7.9 and 7.10 present the performance considering pod reallocation ratio and TTR of DQN-DS and PPO-DS as the number of computing nodes in each cluster decreases from 20 to 2. As visible in Figure 7.9, which shows the statistical distributions of the pod reallocation ratio gathered in 10 different tests, it is easy to note that both



(a) PPO-DS.



(b) DQN-DS.

Figure 7.10: Statistical distribution of the TTR of service requests collected during 10 simulations by decreasing the number of nodes from 20 to 2.

methods already achieve pod reallocation around 80% with only 6 nodes per cluster and reach full reallocation in most runs when 8 nodes per cluster are available. However, PPO-DS seems to show a more deterministic approach compared to DQN-DS. As a result, it tends to repeatedly select the same nodes as potential destinations for the evicted pods, with a rapid consumption of their resources. This likely leads to a deterioration of the pod reallocation ratio (visible in the 20 nodes per cluster case in Figure 7.9a), as PPO-DS continues to select nodes that have already run out of resources. In contrast, DQN-DS adapts more effectively to low and high availability, maintaining

consistently strong reallocation results. Figure 7.10 illustrates the TTR statistical distribution demonstrating a fast convergence to values between 0 and 1 second. Results with 2 nodes per cluster are omitted from these results since they were out of scale, with a very poor node reallocation and a consequent increase in TTR by tens of seconds. For the other node settings, TTR remains steady in a range of 0 to 1 second, even when only 40% of the pods are successfully reallocated (e.g., the case of DQN-DS with 4 nodes per cluster). Finally, the importance of these results is to allow service providers to quantify the minimum number of computing nodes required to run an application with some resilience against unexpected faults. In this case, a good configuration for the target workload described in Table 7.6 is to rent at least 6 nodes per cluster.

7.2 Chapter Summary

Since its initial development at Netflix, CE has evolved to incorporate new concepts and solutions adopted both in industry and academia over the years. By intentionally introducing various types of fault into the system, such as increased network latency or sudden shutdowns, CE enables adopters to analyze the resiliency of the system and detect potential improvements [178], [259]. Moreover, to respond more quickly and efficiently to the rapid growth in market demands, the majority of organizations are increasingly shifting their approach toward new paradigms such as cloud-native ones or Fog Computing to handle domains like IoT and Smart Cities [260]. Several tools have emerged to ease the application of chaos engineering to complex IT systems, leveraging microservice and container-based applications deployed on K8s. However, applying such tools requires several phases to be put into practice, from defining a steady state to establishing an effective response plan if something goes wrong. To ease the application of CE to improve the resilience of K8s applications, this chapter presented a smart scheduler for K8s called TELKA: a Twin-Enhanced Learning for K8s Applications, which combines CE, DT, and RL methodologies to mitigate the effects of computing and network faults. In a first experimentation, TELKA was evaluated in a complex CC scenario in which computing and network faults have been injected to undermine the performance of an image-recognition application deployed on K8s. Initial results highlight the efficacy of the proposal, demonstrating how a CE strategy, when applied to an accurate DT modeling, can bring many benefits. It not only facilitates the testing of IT systems robustness, but can also be leveraged to develop tools to make them more resilient to failures.

Although TELKA showed promising results in reallocating evicted pods, its preliminary implementations suffered from scalability issues, as the RL agent could only effectively operate on scenarios with the same number of nodes seen during training. Therefore, a second architecture based on DS was designed to enhance the performance of TELKA and the resilience of K8s-based applications through the application of CE

methodologies. With the implementation of these novel algorithms and the introduction of DS in traditional PPO and DQN architectures provided by online libraries, TELKA generalization capabilities have been evaluated in a CC scenario already defined in its previous implementation. Outcomes emphasize the validity of this paradigm, demonstrating how DS can remarkably improve the performance of the RL approaches, making their decision-making ability less prone to changes in the reference environment and avoiding the need to spend additional time and computing resources in their retraining.

Conclusion and Future Work

This doctoral research has addressed some of the main challenges of modern distributed systems, focusing on the autonomous orchestration of cloud-native applications across the CC. By integrating DT modeling, RL, and CI into a unified methodological framework, this work has shown how data-driven and simulation-enhanced approaches can substantially improve the efficiency, adaptability, and resilience of orchestration strategies for heterogeneous computing environments.

The main contributions of this thesis are summarized as follows:

- KT: a DT-based framework designed to emulate large-scale K8s deployments. KT enables accurate what-if analyses, allowing the evaluation and optimization of orchestration policies without impacting production environments. Through MDN modeling and simulation-based inference, KT achieves precise characterization of microservice response times, thus validating the feasibility of DT-driven decision support for service management in the CC.

- Comparative analysis and hybridization of RL and CI methodologies in the CC: A systematic evaluation was conducted on the applicability, strengths, and limitations of various RL and CI techniques, including DQN, PPO, GA, PSO, and GWO, for service orchestration in CC environments. The results demonstrated that while RL shows significant performance in long-term decision-making and adaptation to dynamic conditions, CI-based metaheuristics achieve faster convergence and are less sensitive to non-stationary optimization conditions. Based on the comparative findings, the thesis also proposed hybrid methodologies integrating RL agents with CI-based optimization layers. These hybrid approaches leverage RL to accelerate metaheuristics space search and hot restart, thus improving both convergence speed and stability in dynamic CC environments.

- gym-multi-k8s and HephaestusForge: two RL-based environments designed for orchestrating microservice deployments across multi-cluster K8s infrastructures. These frameworks employ DS architectures to achieve generalization across varying cluster configurations. The results demonstrated that RL agents can learn efficient service placement policies that balance latency, cost, and fairness, outperforming heuristic-based strategies in dynamic CC conditions.

- TELKA: an RL scheduler that leverages DT simulations and CE to improve system fault tolerance. By interacting with a virtualized twin instead of a physical deployment, TELKA learns to detect and mitigate node failures and network anomalies, thus reducing recovery time and operational overhead.

In summary, the thesis advances the state-of-the-art in autonomous orchestration for the CC, bridging the gap between simulation fidelity and intelligent decision-making. The proposed frameworks demonstrate that coupling DT–based simulation with learning-driven optimization can yield orchestration mechanisms that are efficient and resilient, paving the way toward fully self-adaptive service management in next-generation cloud-native systems.

Although the results achieved in this work are promising, several directions remain open for further exploration. For example, incorporating MO-RL will allow agents to learn dynamic trade-offs and offer Pareto-optimal solutions that can adapt to evolving operational priorities. In this way, future orchestrators should simultaneously optimize multiple performance criteria (such as latency, cost, energy consumption, and reliability) without reducing them to a single scalar objective. Another key limitation of current traditional RL systems is the need to retrain when the environment shows frequent drastic changes and focuses on finding a solution, rather than treating learning as an endless adaptation. Integrating continual learning and meta-learning paradigms would enable orchestrators to retain knowledge across tasks and adapt incrementally to evolving workloads, infrastructure conditions, or policies in the CC.

Finally, as the CC increasingly spans geographically distributed and heterogeneous infrastructures, orchestration will benefit from decentralized coordination mechanisms. Extending this work toward Federated RL could allow local agents to collaboratively learn orchestration policies while preserving data privacy and reducing communication overhead.

Awards, Invitations, and Notable Scientific Contributions

During the doctoral studies, the author had the opportunity to contribute to the scientific community through a series of research outputs, collaborations, and dissemination activities. Several of these efforts were acknowledged by the academic community, resulting in awards, speech invitations, and other forms of recognition. This section provides a brief overview of the most significant achievements obtained throughout the Ph.D. program.

7.2.1 Awards and Distinction

Among awards and distinctions, the most notable include:

- IFIP-supported 19th International Conference on Network and Service Management (CNSM 2023) Best Paper Award, recognizing L. Manca, D. Borsatti, F. Poltronieri, *et al.*, "Characterization of microservice response time in kubernetes: A mixture density network approach,"
- IEEE/IFIP NOMS 2024 Student Travel Grant
- 2024 CNOM Best Paper Award, recognizing L. Manca, D. Borsatti, F. Poltronieri, *et al.*, "Characterization of microservice response time in kubernetes: A mixture density network approach,"
- 2025 Future Generation Computer Systems (FGCS) Spring Editor's Choice Paper, recognizing J. Santos, M. Zaccarini, F. Poltronieri, *et al.*, "Hephaestusforge: Optimal microservice deployment across the compute continuum via reinforcement learning,"
- IEEE/IFIP NOMS 2025 Student Travel Grant

These distinctions reflect both the scientific relevance of the work conducted and the broader interest it generated within the research community.

7.2.2 Invited Presentations and Talks

This research activity also led to several invitations to present at national and international venues. In particular:

- 3rd IEEE/IFIP Workshop on Technologies for Network Twins (TNT 2024), invitation as a guest speaker to present and discuss the KubeTwin framework.
- 16th International Conference on Network of the Future (NoF 2025), invited as a tutorial speaker to present the proposal "Reinforcement Learning vs. Metaheuristics for Adaptive Orchestration in the Cloud Continuum". Coauthors: Filippo Poltronieri (University of Ferrara), Josè Pedro Pereira dos Santos (Ghent University)

These invitations provided valuable opportunities to disseminate findings, engage in inspiring discussions, and receive feedback from experts in the field.

7.2.3 Notable Scientific Contributions and Collaborations

In addition to formal recognitions, this doctoral work contributed to a number of significant scientific initiatives. These include:

- Development of the KubeTwin framework, available as open-source on Github[3] and on its related website [261].
- Development of the gym-multi-k8s, available as open-source on Github[4].
- Development of the HephaestusForge framework, available as open-source on Github[5].

Overall, the recognitions and contributions summarized in this section highlight the engagement, dedication, and scientific commitment that characterized the doctoral journey.

[3] https://github.com/DSG-UniFE/KubeTwin
[4] https://github.com/mattiazaccarini/multiClusterGentFe
[5] https://github.com/jpedro1992/HephaestusForge

References

[1] S. Moreschini, F. Pecorelli, X. Li, S. Naz, D. Hästbacka, and D. Taibi, "Cloud continuum: The definition," *IEEE Access*, vol. 10, pp. 131 876–131 886, 2022. DOI: 10.1109/ACCESS.2022.3229185.

[2] R. Cavicchioli, R. Martoglia, and M. Verucchi, "A novel real-time edge-cloud big data management and analytics framework for smart cities," *JUCS - Journal of Universal Computer Science*, vol. 28, no. 1, pp. 3–26, 2022, ISSN: 0948-695X. DOI: 10.3897/jucs.71645.

[3] D. Kimovski, R. Mathá, J. Hammer, N. Mehran, H. Hellwagner, and R. Prodan, "Cloud, fog, or edge: Where to compute?" *IEEE Internet Computing*, vol. 25, no. 4, pp. 30–36, 2021. DOI: 10.1109/MIC.2021.3050613.

[4] V. Chang, L. Golightly, P. Modesti, *et al.*, "A survey on intrusion detection systems for fog and cloud computing," *Future Internet*, vol. 14, no. 3, 2022, ISSN: 1999-5903. DOI: 10.3390/fi14030089.

[5] L. Bittencourt, R. Immich, R. Sakellariou, *et al.*, "The internet of things, fog and cloud continuum: Integration and challenges," *Internet of Things*, vol. 3-4, pp. 134–155, 2018, ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2018.09.005.

[6] J. Santos, M. Zaccarini, F. Poltronieri, *et al.*, "Efficient microservice deployment in kubernetes multi-clusters through reinforcement learning," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, 2024, pp. 1–9. DOI: 10.1109/NOMS59830.2024.10575912.

[7] W. Jiang, B. Han, M. A. Habibi, and H. D. Schotten, "The road towards 6g: A comprehensive survey," *IEEE Open Journal of the Communications Society*, vol. 2, pp. 334–366, 2021.

[8] A. Gupta and R. K. Jha, "A survey of 5g network: Architecture and emerging technologies," *IEEE access*, vol. 3, pp. 1206–1232, 2015.

[9] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Transactions on Cloud Computing*, vol. 7, no. 3, pp. 677–692, 2017.

[10] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 155–158, 2018.

[11] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Fog computing: Enabling the management and orchestration of smart city applications in 5g networks," *Entropy*, vol. 20, no. 1, p. 4, 2017.

[12] D. Zhao, G. Sun, D. Liao, S. Xu, and V. Chang, "Mobile-aware service function chain migration in cloud–fog computing," *Future Generation Computer Systems*, vol. 96, pp. 591–604, 2019.

[13] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE access*, vol. 8, pp. 85 714–85 728, 2020.

[14] G. Sun, Y. Li, Y. Li, D. Liao, and V. Chang, "Low-latency orchestration for workflow-oriented service function chain in edge computing," *Future Generation Computer Systems*, vol. 85, pp. 116–128, 2018.

[15] D. Balouek-Thomert, E. G. Renart, A. R. Zamani, A. Simonet, and M. Parashar, "Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1159–1174, 2019.

[16] P. Bellavista, N. Bicocchi, M. Fogli, C. Giannelli, M. Mamei, and M. Picone, "Exploiting microservices and serverless for digital twins in the cloud-to-edge continuum," *Future Generation Computer Systems*, vol. 157, pp. 275–287, 2024.

[17] C. Badue, R. Guidolini, R. V. Carneiro, *et al.*, "Self-driving cars: A survey," *Expert Systems with Applications*, vol. 165, p. 113 816, 2021.

[18] D. C. Nguyen, M. Ding, P. N. Pathirana, *et al.*, "6g internet of things: A comprehensive survey," *IEEE Internet of Things Journal*, 2021.

[19] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2557–2589, 2021.

[20] Z. Zhong and R. Buyya, "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–24, 2020.

[21] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. Leung, "Tailored learning-based scheduling for kubernetes-oriented edge-cloud system," in *IEEE INFOCOM 2021-IEEE conference on computer communications*, IEEE, 2021, pp. 1–10.

[22] J. B. Ssemakula, J.-L. Gorricho, G. Kibalya, and J. Serrat-Fernandez, "An artificial intelligence strategy for the deployment of future microservice-based applications in 6g networks," *Neural Computing and Applications*, vol. 36, no. 18, pp. 10 971–10 997, 2024.

[23] W. Fan, S. Li, J. Liu, Y. Su, F. Wu, and Y. Liu, "Joint task offloading and resource allocation for accuracy-aware machine-learning-based iiot applications," *IEEE Internet of Things Journal*, vol. 10, no. 4, pp. 3305–3321, 2023. DOI: 10.1109/JIOT.2022.3181990.

[24] J. Santos, P. Maniotis, C. Wang, *et al.*, "Evaluating the network effects of orchestration strategies for ai workloads in modern data centers," in *Accepted for Publication at 2025 IEEE Conference on Network Softwarization (NetSoft)*, IEEE, 2025, pp. 1–9.

[25] J. Fields, K. Chovanec, and P. Madiraju, "A survey of text classification with transformers: How wide? how large? how long? how accurate? how expensive? how safe?" *IEEE Access*, vol. 12, pp. 6518–6531, 2024. DOI: 10.1109/ACCESS.2024.3349952.

[26] S. Dustdar, V. C. Pujol, and P. K. Donta, "On distributed computing continuum systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 4, pp. 4092–4105, 2023. DOI: 10.1109/TKDE.2022.3142856.

[27] J. Santos, M. Zaccarini, F. Poltronieri, *et al.*, "Hephaestusforge: Optimal microservice deployment across the compute continuum via reinforcement learning," *Future Generation Computer Systems*, vol. 166, 2025. DOI: 10.1016/j.future.2024.107680.

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

[29] S. Peng, H. Wang, and Q. Yu, "Multi-clusters adaptive brain storm optimization algorithm for qos-aware service composition," *IEEE Access*, vol. 8, pp. 48 822–48 835, 2020. DOI: 10.1109/ACCESS.2020.2979892.

[30] J. Yang, J. Zhang, C. Ma, H. Wang, J. Zhang, and G. Zheng, "Deep learning-based edge caching for multi-cluster heterogeneous networks," *Neural Computing and Applications*, vol. 32, no. 19, pp. 15 317–15 328, 2020. DOI: 10.1007/s00521-019-04040-z.

[31] *Amazon elastic kubernetes service documentation*, https://docs.aws.amazon.com/eks/, [Online; accessed 16-October-2025].

[32] *Platform 9*, https://platform9.com/, [Online; accessed 16-October-2025].

[33] L. Xu, G. Yu, and Y. Jiang, "Energy-efficient resource allocation in single-cell ofdma systems: Multi-objective approach," *IEEE Transactions on Wireless Communications*, vol. 14, no. 10, pp. 5848–5858, 2015. DOI: 10.1109/TWC.2015.2443104.

[34] M. Ito, F. He, and E. Oki, "Robust optimization for probabilistic protection with multiple types of resources in cloud," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, 2020, pp. 1–6. DOI: 10.1109/CloudNet51028.2020.9335810.

[35] A. G. Papidas and G. C. Polyzos, "Self-organizing networks for 5g and beyond: A view from the top," *Future Internet*, vol. 14, no. 3, 2022, ISSN: 1999-5903. DOI: 10.3390/fi14030095.

[36] D. Silver, S. Singh, D. Precup, and R. S. Sutton, "Reward is enough," *Artificial Intelligence*, vol. 299, p. 103 535, 2021, ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2021.103535.

[37] F. Wei, G. Feng, Y. Sun, Y. Wang, and Y.-C. Liang, "Dynamic network slice reconfiguration by exploiting deep reinforcement learning," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–6. DOI: 10.1109/ICC40277.2020.9148848.

[38] P. T. A. Quang, Y. Hadjadj-Aoul, and A. Outtagarts, "A deep reinforcement learning approach for vnf forwarding graph embedding," *IEEE Transactions on Network and Service Management*, vol. 16, no. 4, pp. 1318–1331, 2019. DOI: 10.1109/TNSM.2019.2947905.

[39] A. Kaur and K. Kumar, "Energy-efficient resource allocation in cognitive radio networks under cooperative multi-agent model-free reinforcement learning schemes," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1337–1348, 2020. DOI: 10.1109/TNSM.2020.3000274.

[40] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Reinforcement learning for service function chain allocation in fog computing," in *Communication Networks and Service Management in the Era of Artificial Intelligence and Machine Learning*. John Wiley Sons, Ltd, 2021, ch. 7, pp. 147–173, ISBN: 9781119675525. DOI: https://doi.org/10.1002/9781119675525.ch7.

[41] J. Alonso, L. Orue-Echevarria, E. Osaba, *et al.*, "Optimization and prediction techniques for self-healing and self-learning applications in a trustworthy cloud continuum," *Information*, vol. 12, no. 8, 2021, ISSN: 2078-2489. DOI: 10.3390/info12080308.

[42] X. Ji, Y. Zhang, D. Gong, X. Sun, and Y. Guo, "Multisurrogate-assisted multitasking particle swarm optimization for expensive multimodal problems," *IEEE Transactions on Cybernetics*, pp. 1–15, 2021. DOI: 10.1109/TCYB.2021.3123625.

[43] D. Yazdani, R. Cheng, D. Yazdani, J. Branke, Y. Jin, and X. Yao, "A survey of evolutionary continuous dynamic optimization over two decades—part a," *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 4, pp. 609–629, 2021. DOI: 10.1109/TEVC.2021.3060014.

[44] D. Yazdani, R. Cheng, D. Yazdani, J. Branke, Y. Jin, and X. Yao, "A survey of evolutionary continuous dynamic optimization over two decades—part b," *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 4, pp. 630–650, 2021. DOI: 10.1109/TEVC.2021.3060012.

[45] C. Canali, C. Gazzotti, R. Lancellotti, and F. Schena, "Placement of iot microservices in fog computing systems: A comparison of heuristics," *Algorithms*, vol. 16, no. 9, 2023, ISSN: 1999-4893. DOI: 10.3390/a16090441.

[46] J. Santos, T. Wauters, B. Volckaert, and F. D. Turck, "Resource provisioning in fog computing through deep reinforcement learning," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021, pp. 431–437.

[47] J. Singh, S. Verma, Y. Matsuo, F. Fossati, and G. Fraysse, "Autoscaling packet core network functions with deep reinforcement learning," in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–6. DOI: 10.1109/NOMS56928.2023.10154312.

[48] J. C. Guevara, R. d. S. Torres, L. F. Bittencourt, and N. L. S. da Fonseca, "Qos-aware task scheduling based on reinforcement learning for the cloud-fog continuum," in *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, 2022, pp. 2328–2333. DOI: 10.1109/GLOBECOM48099.2022.10001644.

[49] J. Gómez-delaHiz, J. L. Herrera, D. Scotece, *et al.*, "Evolutionary computation for latency minimization in sdn microservice architectures," in *ICC 2024 - IEEE International Conference on Communications*, 2024, pp. 171–176. DOI: 10.1109/ICC51166.2024.10622476.

[50] M. Asim, Y. Wang, K. Wang, and P.-Q. Huang, "A review on computational intelligence techniques in cloud and edge computing," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 4, no. 6, pp. 742–763, 2020. DOI: 10.1109/TETCI.2020.3007905.

[51] S. Laso, L. Tore-Gálvez, J. Berrocal, C. Canal, and J. M. Murillo, "Deploying digital twins over the cloud-to-thing continuum," in *2023 IEEE Symposium on Computers and Communications (ISCC)*, 2023, pp. 1–6. DOI: 10.1109/ISCC58397.2023.10218052.

[52] I. Errandonea, S. Beltrán, and S. Arrizabalaga, "Digital twin for maintenance: A literature review," *Computers in Industry*, vol. 123, p. 103 316, 2020, ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2020.103316. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166361520305509.

[53] P. Bellavista, C. Giannelli, M. Mamei, M. Mendula, and M. Picone, "Application-driven network-aware digital twin management in industrial edge environments," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 11, pp. 7791–7801, 2021. DOI: 10.1109/TII.2021.3067447.

[54] L. Zhao, G. Han, Z. Li, and L. Shu, "Intelligent digital twin-based software-defined vehicular networks," *IEEE Network*, vol. 34, no. 5, pp. 178–184, 2020. DOI: 10.1109/MNET.011.1900587.

[55] M. Balogh, A. Földvári, and P. Varga, "Digital twins in industry 5.0: Challenges in modeling and communication," in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–6. DOI: 10.1109/NOMS56928.2023.10154424.

[56] Y. Wu, K. Zhang, and Y. Zhang, "Digital twin networks: A survey," *IEEE Internet of Things Journal*, vol. 8, no. 18, pp. 13 789–13 804, 2021. DOI: 10.1109/JIOT.2021.3079510.

[57] W. Cerroni, L. Foschini, G. Y. Grabarnik, *et al.*, "Bdmaas+: Business-driven and simulation-based optimization of it services in the hybrid cloud," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 322–337, 2022. DOI: 10.1109/TNSM.2021.3110139.

[58] F. Poltronieri, M. Tortonesi, and C. Stefanelli, "Chaostwin: A chaos engineering and digital twin approach for the design of resilient it services," in *2021 17th International Conference on Network and Service Management (CNSM)*, 2021, pp. 234–238. DOI: 10.23919/CNSM52442.2021.9615519.

[59] W. Wang, L. Tang, C. Wang, and Q. Chen, "Real-time analysis of multiple root causes for anomalies assisted by digital twin in nfv environment," *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 905–921, 2022. DOI: 10.1109/TNSM.2022.3151249.

[60] M. Zhu, R. Kang, F. He, and E. Oki, "Implementation of backup resource management controller for reliable function allocation in kubernetes," in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, 2021, pp. 360–362. DOI: 10.1109/NetSoft51509.2021.9492724.

[61] M. Fogli, T. Kudla, B. Musters, *et al.*, "Performance evaluation of kubernetes distributions (k8s, k3s, kubeedge) in an adaptive and federated cloud infrastructure for disadvantaged tactical networks," in *2021 International Conference on Military Communication and Information Systems (ICMCIS)*, IEEE, 2021, pp. 1–7.

[62] *Karmada documentation*. https://karmada.io/, [Online; accessed 10-September-2025].

[63] R. Marler and J. Arora, "Survey of multi-objective optimization methods for engineering," *Structural and Multidisciplinary Optimization*, vol. 26, pp. 369–395, Apr. 2004. DOI: 10.1007/s00158-003-0368-6.

[64] C. F. Hayes, R. Rădulescu, E. Bargiacchi, *et al.*, "A practical guide to multi-objective reinforcement learning and planning," *Autonomous Agents and Multi-Agent Systems*, vol. 36, no. 1, p. 26, 2022.

[65] M. De Donno, K. Tange, and N. Dragoni, "Foundations and evolution of modern computing paradigms: Cloud, iot, edge, and fog," *IEEE Access*, vol. 7, pp. 150 936–150 948, 2019. DOI: 10.1109/ACCESS.2019.2947652.

[66] A. Taherkordi, F. Zahid, Y. Verginadis, and G. Horn, "Future cloud systems design: Challenges and research directions," *IEEE Access*, vol. 6, pp. 74 120–74 150, 2018. DOI: 10.1109/ACCESS.2018.2883149.

[67] A. Koohang, C. S. Sargent, J. H. Nord, and J. Paliszkiewicz, "Internet of things (iot): From awareness to continued use," *International Journal of Information Management*, vol. 62, p. 102 442, 2022, ISSN: 0268-4012. DOI: https://doi.org/10.1016/j.ijinfomgt.2021.102442.

[68] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22–32, 2014. DOI: 10.1109/JIOT.2014.2306328.

[69] H. Xu, W. Yu, D. Griffith, and N. Golmie, "A survey on industrial internet of things: A cyber-physical systems perspective," *IEEE Access*, vol. 6, pp. 78 238–78 259, 2018. DOI: 10.1109/ACCESS.2018.2884906.

[70] Y. Yin, Y. Zeng, X. Chen, and Y. Fan, "The internet of things in healthcare: An overview," *Journal of Industrial Information Integration*, vol. 1, pp. 3–13, 2016, ISSN: 2452-414X. DOI: https://doi.org/10.1016/j.jii.2016.03.004.

[71] H. Lu, Q. Liu, D. Tian, Y. Li, H. Kim, and S. Serikawa, "The cognitive internet of vehicles for autonomous driving," *IEEE Network*, vol. 33, no. 3, pp. 65–73, 2019. DOI: 10.1109/MNET.2019.1800339.

[72] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, and T. Zhou, "A survey on edge computing systems and tools," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1537–1562, 2019. DOI: 10.1109/JPROC.2019.2920341.

[73] R. K. Naha, S. Garg, D. Georgakopoulos, *et al.*, "Fog computing: Survey of trends, architectures, requirements, and research directions," *IEEE Access*, vol. 6, pp. 47 980–48 009, 2018. DOI: 10.1109/ACCESS.2018.2866491.

[74] "Ieee standard for adoption of openfog reference architecture for fog computing," *IEEE Std 1934-2018*, pp. 1–176, 2018. DOI: 10.1109/IEEESTD.2018.8423800.

[75] M. Tortonesi, "The compute continuum: Trends and challenges," *Computer*, vol. 58, no. 3, pp. 105–108, 2025. DOI: 10.1109/MC.2024.3520255.

[76] J. McCarthy, "Time-sharing computer systems," in *Computers and the World of the Future*, M. Greenberger, Ed., Cambridge, MA: MIT Press, 1962, pp. 220–248.

[77] R. J. Creasy, "The origin of the vm/370 time-sharing system," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, 1981. DOI: 10.1147/rd.255.0483.

[78] R. Venkateswaran, "Virtual private networks," *IEEE Potentials*, vol. 20, no. 1, pp. 11–15, 2001. DOI: 10.1109/45.913204.

[79] *Amazon ec2*, https://aws.amazon.com/ec2/, [Online; accessed 27-August-2025].

[80] *Amazon s3*, https://aws.amazon.com/s3/, [Online; accessed 27-August-2025].

[81] *Who coined 'cloud computing'?* https://www.technologyreview.com/2011/10/31/257406/who-coined-cloud-computing/, [Online; accessed 27-August-2025].

[82] P. Gupta, A. Seetharaman, and J. R. Raj, "The usage and adoption of cloud computing by small and medium businesses," *International Journal of Information Management*, vol. 33, no. 5, pp. 861–874, 2013, ISSN: 0268-4012. DOI: https://doi.org/10.1016/j.ijinfomgt.2013.07.001.

[83] *Docker*, https://www.docker.com/, [Online; accessed 27-August-2025].

[84] *Kubernetes*, https://kubernetes.io/, [Online; accessed 27-August-2025].

[85] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *IEEE Internet Computing*, vol. 13, no. 5, pp. 14–22, 2009. DOI: 10.1109/MIC.2009.119.

[86] J. Hong, T. Dreibholz, J. A. Schenkel, and J. A. Hu, "An overview of multi-cloud computing," in *Web, Artificial Intelligence and Network Applications*, L. Barolli, M. Takizawa, F. Xhafa, and T. Enokido, Eds., Springer International Publishing, 2019, pp. 1055–1068.

[87] *The little-known story of the first iot device*, https://www.ibm.com/think/topics/iot-first-device, [Online; accessed 28-August-2025].

[88] J. Romkey, "Toast of the iot: The 1990 interop internet toaster," *IEEE Consumer Electronics Magazine*, vol. 6, no. 1, pp. 116–119, 2017. DOI: 10.1109/MCE.2016.2614740.

[89] *The trojan room coffee pot, a (non-technical) biography*, https://www.cl.cam.ac.uk/coffee/qsf/coffee.html, [Online; accessed 28-August-2025].

[90] *Usno navstar global positioning system*, https://web.archive.org/web/20110126200746/http://tycho.usno.navy.mil/gpsinfo.html, [Online; accessed 28-August-2025].

[91] *Internet protocol, version 6 (ipv6) specification*, https://www.ietf.org/rfc/rfc2460.txt, [Online; accessed 28-August-2025].

[92] T. Kramp, R. van Kranenburg, and S. Lange, "Introduction to the internet of things," in *Enabling Things to Talk: Designing IoT solutions with the IoT Architectural Reference Model*, A. Bassi, M. Bauer, M. Fiedler, *et al.*, Eds. Springer Berlin Heidelberg, 2013, ISBN: 978-3-642-40403-0. DOI: 10.1007/978-3-642-40403-0_1.

[93] *Apple reinvents the phone with iphone*, https://www.apple.com/newsroom/2007/01/09Apple-Reinvents-the-Phone-with-iPhone/, [Online; accessed 01-September-2025].

[94] *Internet of things 2008, international conference for industry and academia*, https://iot-conference.org/iot2008/, [Online; accessed 01-September-2025].

[95] J. Wang, M. K. Lim, C. Wang, and M.-L. Tseng, "The evolution of the internet of things (iot) over the past 20 years," *Computers Industrial Engineering*, vol. 155, p. 107 174, 2021, ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2021.107174.

[96] M. Ford and W. Palmer, "Alexa, are you listening to me? an analysis of alexa voice service network traffic," *Personal and Ubiquitous Computing*, vol. 23, no. 1, pp. 67–79, 2019, ISSN: 1617-4917. DOI: 10.1007/s00779-018-1174-x.

[97] G. G. Samatas, S. S. Moumgiakmas, and G. A. Papakostas, "Predictive maintenance - bridging artificial intelligence and iot," in *2021 IEEE World AI IoT Congress (AIIoT)*, 2021, pp. 0413–0419. DOI: 10.1109/AIIoT52608.2021.9454173.

[98] D.-H. Shin, "Conceptualizing and measuring quality of experience of the internet of things: Exploring how quality is perceived by users," *Information Management*, vol. 54, no. 8, pp. 998–1011, 2017, ISSN: 0378-7206. DOI: https://doi.org/10.1016/j.im.2017.02.006.

[99] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, "On blockchain and its integration with iot. challenges and opportunities," *Future Generation Computer Systems*, vol. 88, pp. 173–190, 2018, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.05.046.

[100] D. W. Matolak, "V2v communication channels: State of knowledge, new results, and what's next," in *Communication Technologies for Vehicles*, M. Berbineau, M. Jonsson, J.-M. Bonnin, *et al.*, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–21, ISBN: 978-3-642-37974-1.

[101] J. E. Siegel, D. C. Erb, and S. E. Sarma, "A survey of the connected vehicle landscape—architectures, enabling technologies, applications, and development areas," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 8, pp. 2391–2406, 2018. DOI: 10.1109/TITS.2017.2749459.

[102] T. Malche, P. Maheshwary, and R. Kumar, "Environmental monitoring system for smart city based on secure internet of things (iot) architecture," *Wireless Personal Communications*, vol. 107, no. 4, pp. 2143–2172, 2019, ISSN: 1572-834X. DOI: 10.1007/s11277-019-06376-0.

[103] Z. Chen, C. Sivaparthipan, and B. Muthu, "Iot based smart and intelligent smart city energy optimization," *Sustainable Energy Technologies and Assessments*, vol. 49, p. 101 724, 2022, ISSN: 2213-1388. DOI: https://doi.org/10.1016/j.seta.2021.101724.

[104] L. F. Bittencourt, R. Rodrigues-Filho, J. Spillner, *et al.*, "The computing continuum: Past, present, and future," *Computer Science Review*, vol. 58, p. 100 782, 2025, ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2025.100782.

[105] *European commission - european industrial technology roadmap for the next generation cloud-edge offering*, [Online; accessed 03-September-2025].

[106] F. Poltronieri, C. Stefanelli, M. Tortonesi, and M. Zaccarini, "Reinforcement learning vs. computational intelligence: Comparing service management approaches for the cloud continuum," *Future Internet*, vol. 15, no. 11, 2023. DOI: 10.3390/fi15110359.

[107] A. Gholami, K. Rao, W.-P. Hsiung, O. Po, M. Sankaradas, and S. Chakradhar, "Roma: Resource orchestration for microservices-based 5g applications," in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 2022, pp. 1–9. DOI: 10.1109/NOMS54207.2022.9789821.

[108] P. Pereira, C. Melo, J. Araujo, J. Dantas, V. Santos, and P. Maciel, "Availability model for edge-fog-cloud continuum: An evaluation of an end-to-end infrastructure of intelligent traffic management service," *The Journal of Supercomputing*, vol. 78, no. 3, pp. 4421–4448, 2022, ISSN: 1573-0484. DOI: 10.1007/s11227-021-04033-7.

[109] M. Verkerken, J. Santos, L. D'Hooge, T. Wauters, B. Volckaert, and F. De Turck, "Chronosguard: A hierarchical machine learning intrusion detection system for modern clouds," in *2024 20th International Conference on Network and Service Management (CNSM)*, 2024, pp. 1–9. DOI: 10.23919/CNSM62983.2024.10814370.

[110] C. Barz, E. Cramer, R. Fronteddu, *et al.*, "Enabling adaptive communications at the tactical edge," in *MILCOM 2022 - 2022 IEEE Military Communications Conference (MILCOM)*, 2022, pp. 1038–1044. DOI: 10.1109/MILCOM55135.2022.10017459.

[111] R. Kruse, C. Borgelt, C. Braune, *et al.*, *Computational Intelligence: A Methodological Introduction*, 2nd. Springer Publishing Company, Incorporated, 2016, ISBN: 1447172949.

[112] D. Yazdani, M. N. Omidvar, R. Cheng, J. Branke, T. T. Nguyen, and X. Yao, "Benchmarking continuous dynamic optimization: Survey and generalized test suite," *IEEE Transactions on Cybernetics*, vol. 52, no. 5, pp. 3380–3393, 2022.

[113] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: Past, present, and future," *Multimedia Tools and Applications*, vol. 80, no. 5, pp. 8091–8126, 2021, ISSN: 1573-7721. DOI: 10.1007/s11042-020-10139-6.

[114] T. Blickle and L. Thiele, "A comparison of selection schemes used in evolutionary algorithms," *Evolutionary Computation*, vol. 4, no. 4, pp. 361–394, 1996, ISSN: 1063-6560. DOI: 10.1162/evco.1996.4.4.361.

[115] M. Tortonesi and L. Foschini, "Business-driven service placement for highly dynamic and distributed cloud systems," *IEEE Transactions on Cloud Computing*, vol. 6, no. 4, pp. 977–990, 2018. DOI: 10.1109/TCC.2016.2541141.

[116] T. M. Shami, A. A. El-Saleh, M. Alswaitti, Q. Al-Tashi, M. A. Summakieh, and S. Mirjalili, "Particle swarm optimization: A comprehensive survey," *IEEE Access*, vol. 10, pp. 10 031–10 061, 2022. DOI: 10.1109/ACCESS.2022.3142859.

[117] A. Ratnaweera, S. Halgamuge, and H. Watson, "Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 3, pp. 240–255, 2004.

[118] "Population size in particle swarm optimization," *Swarm and Evolutionary Computation*, vol. 58, p. 100 718, 2020, ISSN: 2210-6502. DOI: https://doi.org/10.1016/j.swevo.2020.100718.

[119] J. Sun, C.-H. Lai, and X.-J. Wu, *Particle Swarm Optimisation: Classical and Quantum Perspectives*, 1st. CRC Press, 2012. DOI: 10.1201/b11579.

[120] S. Yang, M. Wang, and L. jiao, "A quantum particle swarm optimization," in *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*, vol. 1, 2004, 320–324 Vol.1. DOI: 10.1109/CEC.2004.1330874.

[121] W. Fang, J. Sun, Y. Ding, X. Wu, and W. Xu, "A review of quantum-behaved particle swarm optimization," *IETE Technical Review*, vol. 27, no. 4, pp. 336–348, 2010. DOI: 10.4103/0256-4602.64601.

[122] T. Blackwell, "Particle swarm optimization in dynamic environments," in *Evolutionary Computation in Dynamic and Uncertain Environments*, S. Yang, Y.-S. Ong, and Y. Jin, Eds. Springer Berlin Heidelberg, 2007, pp. 29–49, ISBN: 978-3-540-49774-5. DOI: 10.1007/978-3-540-49774-5_2.

[123] T. Blackwell and J. Branke, "Multi-swarm optimization in dynamic environments," in *Applications of Evolutionary Computing*, G. R. Raidl, S. Cagnoni, J. Branke, *et al.*, Eds., Springer Berlin Heidelberg, 2004, pp. 489–500, ISBN: 978-3-540-24653-4.

[124] S. Mirjalili, S. M. Mirjalili, and A. Lewis, "Grey wolf optimizer," *Advances in Engineering Software*, vol. 69, pp. 46–61, 2014, ISSN: 0965-9978. DOI: https://doi.org/10.1016/j.advengsoft.2013.12.007.

[125] C. L. Camacho-Villalón, M. Dorigo, and T. Stützle, "Exposing the grey wolf, moth-flame, whale, firefly, bat, and antlion algorithms: Six misleading optimization techniques inspired by bestial metaphors," *International Transactions in Operational Research*, vol. 30, no. 6, pp. 2945–2971, 2023. DOI: https://doi.org/10.1111/itor.13176.

[126] M. T. J. Spaan, "Partially observable markov decision processes," in *Reinforcement Learning: State-of-the-Art*, M. Wiering and M. van Otterlo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 387–414, ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_12.

[127] R. Singh, A. Gupta, and N. B. Shroff, "Learning in constrained markov decision processes," *IEEE Transactions on Control of Network Systems*, vol. 10, no. 1, pp. 441–453, 2023. DOI: 10.1109/TCNS.2022.3203361.

[128] P. Vamplew, B. J. Smith, J. Källström, *et al.*, "Scalar reward is not enough: A response to silver, singh, precup and sutton (2021)," *Autonomous Agents and Multi-Agent Systems*, vol. 36, no. 2, p. 41, 2022, ISSN: 1573-7454. DOI: 10.1007/s10458-022-09575-5.

[129] J. Richens, D. Abel, A. Bellot, and T. Everitt, *General agents contain world models*, 2025. [Online]. Available: https://arxiv.org/abs/2506.01622.

[130] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992, ISSN: 1573-0565. DOI: 10.1007/BF00992698.

[131] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, ISSN: 1476-4687. DOI: 10.1038/nature14236.

[132] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, vol. 8, no. 3, pp. 293–321, May 1992, ISSN: 1573-0565. DOI: 10.1007/BF00992699.

[133] B. T. Polyak and A. B. Juditsky, "Acceleration of stochastic approximation by averaging," *SIAM Journal on Control and Optimization*, vol. 30, no. 4, pp. 838–855, 1992. DOI: 10.1137/0330046.

[134] M. Sabry and A. M. A. Khalifa, *On the reduction of variance and overestimation of deep q-learning*, 2024. [Online]. Available: https://arxiv.org/abs/1910.05983.

[135] J. F. Cevallos Moreno, R. Sattler, R. P. Caulier Cisterna, L. Ricciardi Celsi, A. Sánchez Rodríguez, and M. Mecella, "Online service function chain deployment for live-streaming in virtualized content delivery networks: A deep reinforcement learning approach," *Future Internet*, vol. 13, no. 11, 2021, ISSN: 1999-5903. DOI: 10.3390/fi13110278.

[136] Y. Fang, C. Huang, Y. Xu, and Y. Li, "Rlxss: Optimizing xss detection model to defend against adversarial attacks based on reinforcement learning," *Future Internet*, vol. 11, no. 8, 2019, ISSN: 1999-5903. DOI: 10.3390/fi11080177.

[137] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15, Lille, France: JMLR.org, 2015, pp. 1889–1897.

[138] J. Achiam, D. Held, A. Tamar, and P. Abbeel, "Constrained policy optimization," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17, Sydney, NSW, Australia: JMLR.org, 2017, pp. 22–31.

[139] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. arXiv: 1707.06347 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1707.06347.

[140] N. Heess, D. TB, S. Sriram, *et al.*, *Emergence of locomotion behaviours in rich environments*, 2017. arXiv: 1707.02286 [cs.AI]. [Online]. Available: https://arxiv.org/abs/1707.02286.

[141] H. Dong, Z. Ding, and S. Zhang, *Deep Reinforcement Learning Fundamentals, Research and Applications: Fundamentals, Research and Applications*. Jan. 2020, ISBN: 978-981-15-4094-3. DOI: 10.1007/978-981-15-4095-0.

[142] M. Zaheer, S. Kottur, S. Ravanbhakhsh, B. Póczos, R. Salakhutdinov, and A. J. Smola, "Deep sets," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Curran Associates Inc., 2017, pp. 3394–3404, ISBN: 9781510860964.

[143] N. D. Cicco, G. F. Pittalà, G. Davoli, *et al.*, "Drl-forch: A scalable deep reinforcement learning-based fog computing orchestrator," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, 2023, pp. 125–133. DOI: 10.1109/NetSoft57336.2023.10175398.

[144] T. Tušar and B. Filipič, "Visualization of pareto front approximations in evolutionary multiobjective optimization: A critical review and the prosection method," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 2, pp. 225–245, 2014.

[145] G. Mavrotas, "Effective implementation of the ϵ-constraint method in multi-objective mathematical programming problems," *Applied Mathematics and Computation*, vol. 213, no. 2, pp. 455–465, 2009, ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2009.03.037.

[146] S. Tamby and D. Vanderpooten, "Enumeration of the nondominated set of multiobjective discrete optimization problems," *INFORMS Journal on Computing*, vol. 33, no. 1, pp. 72–85, 2021.

[147] M. Á. Domínguez-Ríos, F. Chicano, and E. Alba, "Effective anytime algorithm for multiobjective combinatorial optimization problems," *Information Sciences*, vol. 565, pp. 210–228, 2021, ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2021.02.074.

[148] Q. Huangfu and J. J. Hall, "Parallelizing the dual revised simplex method," *Mathematical Programming Computation*, vol. 10, no. 1, pp. 119–142, 2018.

[149] J. D. Schaffer, "Multiple objective optimization with vector evaluated genetic algorithms," in *Proceedings of the 1st International Conference on Genetic Algorithms*, L. Erlbaum Associates Inc., 1985, pp. 93–100, ISBN: 0805804269.

[150] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002. DOI: 10.1109/4235.996017.

[151] N. Srinivas and K. Deb, "Multiobjective optimization using nondominated sorting in genetic algorithms," *Evolutionary Computation*, vol. 2, no. 3, pp. 221–248, 1994. DOI: 10.1162/evco.1994.2.3.221.

[152] J. Blank, K. Deb, and P. C. Roy, "Investigating the normalization procedure of nsga-iii," in *International Conference on Evolutionary Multi-Criterion Optimization*, Springer, 2019, pp. 229–240.

[153] C. Coello, G. Pulido, and M. Lechuga, "Handling multiple objectives with particle swarm optimization," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 3, pp. 256–279, 2004. DOI: 10.1109/TEVC.2004.826067.

[154] M. R. Sierra and C. A. Coello Coello, "Improving pso-based multi-objective optimization using crowding, mutation and ϵ-dominance," in *Evolutionary Multi-Criterion Optimization*, C. A. Coello Coello, A. Hernández Aguirre, and E. Zitzler, Eds., Springer Berlin Heidelberg, 2005, pp. 505–519, ISBN: 978-3-540-31880-4.

[155] S. Sedarous, S. M. El-Gokhy, and E. Sallam, "Multi-swarm multi-objective optimization based on a hybrid strategy," *Alexandria Engineering Journal*, vol. 57, no. 3, pp. 1619–1629, 2018, ISSN: 1110-0168. DOI: https://doi.org/10.1016/j.aej.2017.06.017.

[156] Z.-H. Zhan, J. Li, J. Cao, J. Zhang, H. S.-H. Chung, and Y.-H. Shi, "Multiple populations for multiple objectives: A coevolutionary technique for solving multiobjective optimization problems," *IEEE Transactions on Cybernetics*, vol. 43, no. 2, pp. 445–463, 2013. DOI: 10.1109/TSMCB.2012.2209115.

[157] M. Bendechache, S. Svorobej, P. Takako Endo, and T. Lynn, "Simulating resource management across the cloud-to-thing continuum: A survey and future directions," *Future Internet*, vol. 12, no. 6, 2020, ISSN: 1999-5903. DOI: 10.3390/fi12060095.

[158] F. Poltronieri, C. Stefanelli, N. Suri, and M. Tortonesi, "Value is king: The mecforge deep reinforcement learning solution for resource management in 5g and beyond," *Journal of Network and Systems Management*, vol. 30, no. 4, 2022. DOI: 10.1007/s10922-022-09672-6.

[159] A. Kanervisto, C. Scheller, and V. Hautamäki, *Action space shaping in deep reinforcement learning*, 2020. DOI: 10.48550/ARXIV.2004.00980.

[160] F. Poltronieri, C. Stefanelli, N. Suri, and M. Tortonesi, "Phileas: A simulation-based approach for the evaluation of value-based fog services," in *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, 2018, pp. 1–6. DOI: 10.1109/CAMAD.2018.8514969.

[161] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html.

[162] *Ruby-mhl - a ruby metaheuristics library*, https://github.com/DSG-UniFE/ruby-mhl, [Online; accessed 17-September-2025].

[163] S. Luke, *Essentials of Metaheuristics*, Second. Lulu, 2015, Available for free at http://cs.gmu.edu/~sean/book/metaheuristics/.

[164] F. Poltronieri, M. Tortonesi, A. Morelli, C. Stefanelli, and N. Suri, "Value of information based optimal service fabric management for fog computing," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–9.

[165] M. Giordani, M. Polese, M. Mezzavilla, S. Rangan, and M. Zorzi, "Toward 6g networks: Use cases and technologies," *IEEE Communications Magazine*, vol. 58, no. 3, pp. 55–61, 2020. DOI: 10.1109/MCOM.001.1900411.

[166] J. J. L. Escobar, F. Gil-Castiñeira, and R. P. Díaz Redondo, "Decentralized serverless iot dataflow architecture for the cloud-to-edge continuum," in *2023 26th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2023, pp. 42–49. DOI: 10.1109/ICIN56760.2023.10073502.

[167] M. Tortonesi, M. Govoni, A. Morelli, G. Riberto, C. Stefanelli, and N. Suri, "Taming the iot data deluge: An innovative information-centric service model for fog computing applications," *Future Generation Computer Systems*, vol. 93, pp. 888–902, 2019, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.06.009.

[168] N. Suri, G. Benincasa, R. Lenzi, M. Tortonesi, C. Stefanelli, and L. Sadler, "Exploring value-of-information-based approaches to support effective communications in tactical networks," *IEEE Communications Magazine*, vol. 53, no. 10, pp. 39–45, 2015. DOI: 10.1109/MCOM.2015.7295461.

[169] F. Poltronieri, M. Tortonesi, A. Morelli, C. Stefanelli, and N. Suri, "A value-of-information-based management framework for fog services," *International Journal of Network Management*, vol. 32, no. 1, e2156, 2022. DOI: https://doi.org/10.1002/nem.2156.

[170] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, 2022. DOI: 10.1109/TPDS.2021.3128037.

[171] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, "Osmotic computing: A new paradigm for edge/cloud integration," *IEEE Cloud Computing*, vol. 3, no. 6, pp. 76–83, 2016. DOI: 10.1109/MCC.2016.124.

[172] M. Zaccarini, B. Cantelli, M. Fazio, *et al.*, "Voice: Value-of-information for compute continuum ecosystems," in *2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN)*, 2024, pp. 73–80. DOI: 10.1109/ICIN60470.2024.10494500.

[173] D. Borsatti, W. Cerroni, L. Foschini, *et al.*, "Modeling digital twins of kubernetes-based applications," in *2023 IEEE Symposium on Computers and Communications (ISCC)*, 2023, pp. 219–224. DOI: 10.1109/ISCC58397.2023.10217853.

[174] L. Manca, D. Borsatti, F. Poltronieri, *et al.*, "Characterization of microservice response time in kubernetes: A mixture density network approach," in *2023 19th International Conference on Network and Service Management (CNSM)*, 2023, pp. 1–9. DOI: 10.23919/CNSM59352.2023.10327842.

[175] R. Minerva, G. M. Lee, and N. Crespi, "Digital twin in the iot context: A survey on technical features, scenarios, and architectural models," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1785–1824, 2020. DOI: 10.1109/JPROC.2020.2998530.

[176] *Cloudping - aws latency monitoring*, https://www.cloudping.co/, [Online; accessed 16-September-2025].

[177] S. Laso, L. Martín, J. L. Herrera, J. Galán-Jiménez, J. Berrocal, and J. M. Murillo, "Dantalion: Digital twinning the computing continuum," in *2023 IEEE Globecom Workshops (GC Wkshps)*, 2023, pp. 1303–1306. DOI: 10.1109/GCWkshps58843.2023.10464899.

[178] A. Basiri, N. Behnam, R. de Rooij, *et al.*, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016. DOI: 10.1109/MS.2016.60.

[179] *Kubernetes overview*, https://kubernetes.io/docs/concepts/overview/, [Online; accessed 18-September-2025].

[180] *Kubernetes components*, https://kubernetes.io/docs/concepts/overview/components/, [Online; accessed 18-September-2025].

[181] *What is kubernetes control plane?* https://www.geeksforgeeks.org/devops/what-is-kubernetes-control-plane/, [Online; accessed 18-September-2025].

[182] *Using coredns for service discovery*, https://kubernetes.io/docs/tasks/administer-cluster/coredns/, [Online; accessed 19-September-2025].

[183] *Cloud controller manager*, https://kubernetes.io/docs/concepts/architecture/cloud-controller/, [Online; accessed 19-September-2025].

[184] *Kubernetes self-healing*, https://kubernetes.io/docs/concepts/architecture/self-healing/, [Online; accessed 19-September-2025].

[185] *Kubernetes secrets*, https://kubernetes.io/docs/concepts/configuration/secret/, [Online; accessed 19-September-2025].

[186] *Kubernetes pods*, https://kubernetes.io/docs/concepts/workloads/pods/, [Online; accessed 22-September-2025].

[187] *Kubernetes services*, https://kubernetes.io/docs/concepts/services-networking/service/, [Online; accessed 22-September-2025].

[188] *Kubernetes workloads*, https://kubernetes.io/docs/concepts/workloads/, [Online; accessed 22-September-2025].

[189] E. Stoneman, *Learn Kubernetes in a Month of Lunches*. Manning, Feb. 2021, p. 592, ISBN: 9781617297984.

[190] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015. DOI: 10.1109/MS.2015.11.

[191] N. Dragoni, S. Giallorenzo, A. L. Lafuente, *et al.*, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Cham: Springer International Publishing, 2017, pp. 195–216, ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12.

[192] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019, pp. 351–359. DOI: 10.1109/NETSOFT.2019.8806671.

[193] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, 2016, pp. 20–26. DOI: 10.1109/SmartCloud.2016.18.

[194] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, 2019. DOI: 10.1109/JPROC.2019.2915983.

[195] Y. Siriwardhana, P. Porambage, M. Liyanage, and M. Ylianttila, "A survey on mobile augmented reality with 5g mobile edge computing: Architectures, applications, and technical aspects," *IEEE Communications Surveys Tutorials*, vol. 23, no. 2, pp. 1160–1192, 2021. DOI: 10.1109/COMST.2021.3061981.

[196] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions," *IEEE Communications Surveys Tutorials*, vol. 23, no. 4, pp. 2557–2589, 2021. DOI: 10.1109/COMST.2021.3095358.

[197] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, "Optimal virtual network function placement in multi-cloud service function chaining architecture," *Computer Communications*, vol. 102, pp. 1–16, 2017.

[198] C. Guerrero, I. Lera, and C. Juiz, "Resource optimization of container orchestration: A case study in multi-cloud microservices-based applications," *The Journal of Supercomputing*, vol. 74, no. 7, pp. 2956–2983, 2018.

[199] S. K. Panda, I. Gupta, and P. K. Jana, "Task scheduling algorithms for multi-cloud systems: Allocation-aware approach," *Information Systems Frontiers*, vol. 21, pp. 241–259, 2019.

[200] S. Qin, D. Pi, Z. Shao, Y. Xu, and Y. Chen, "Reliability-aware multi-objective memetic algorithm for workflow scheduling problem in multi-cloud system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 4, pp. 1343–1361, 2023. DOI: 10.1109/TPDS.2023.3245089.

[201] *Kubefed*. https://github.com/kubernetes-retired/kubefed, [Online; accessed 11-September-2025].

[202] *Kubernetes cluster-api*. https://github.com/kubernetes-sigs/cluster-api, [Online; accessed 11-September-2025].

[203] *Open cluster management.* https://open-cluster-management.io/, [Online; accessed 11-September-2025].

[204] B. Burns, J. Beda, and K. Hightower, *Kubernetes: up and running: dive into the future of infrastructure*. O'Reilly Media, 2019.

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

[206] R. Galliera, A. Morelli, R. Fronteddu, and N. Suri, "Marlin: Soft actor-critic based reinforcement learning for congestion control in real networks," in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, IEEE, 2023, pp. 1–10.

[207] G. Brockman, V. Cheung, L. Pettersson, *et al.*, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[208] *Amazon ec2 on-demand pricing*. https://aws.amazon.com/ec2/pricing/on-demand/, [Online; accessed 10-September-2025].

[209] H. Sami, A. Mourad, H. Otrok, and J. Bentahar, "Demand-driven deep reinforcement learning for scalable fog and service placement," *IEEE Transactions on Services Computing*, vol. 15, no. 5, pp. 2671–2684, 2021.

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

[211] V. Mnih, A. P. Badia, M. Mirza, *et al.*, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, PMLR, 2016, pp. 1928–1937.

[212] C.-Y. Tang, C.-H. Liu, W.-K. Chen, and S. D. You, "Implementing action mask in proximal policy optimization (ppo) algorithm," *ICT Express*, vol. 6, no. 3, pp. 200–203, 2020.

[213] M. Aly, F. Khomh, and S. Yacout, "Kubernetes or openshift? which technology best suits eclipse hono iot deployments," in *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*, 2018, pp. 113–120. DOI: 10.1109/SOCA.2018.00024.

[214] J. Lee, S. Kang, and I.-G. Chun, "Miotwins: Design and evaluation of miot framework for private edge networks," in *2021 International Conference on Information and Communication Technology Convergence (ICTC)*, 2021, pp. 1882–1884. DOI: 10.1109/ICTC52510.2021.9621144.

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

[216] J. Kristan, P. Azzoni, L. Römer, S. E. Jeroschewski, and E. Londero, "Evolving the ecosystem: Eclipse arrowhead integrates eclipse iot," in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 2022, pp. 1–6. DOI: 10.1109/NOMS54207.2022.9789922.

[217] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Diktyo: Network-aware scheduling in container-based clouds," *IEEE Transactions on Network and Service Management*, 2023.

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

[219] K. Blesch, O. P. Hauser, and J. M. Jachimowicz, "Measuring inequality beyond the gini coefficient may clarify conflicting findings," *Nature human behaviour*, vol. 6, no. 11, pp. 1525–1536, 2022.

[220] Amazon AWS, *Amazon ec2 reserved instance pricing*. https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/?nc1=h_ls, [Online Accessed on 15-September-2025].

[221] V. Noghin, "Linear scalarization in multi-criterion optimization," *Scientific and Technical Information Processing*, vol. 42, pp. 463–469, 2015.

[222] E. M. Arkin and E. B. Silverberg, "Scheduling jobs with fixed start and end times," *Discrete Applied Mathematics*, vol. 18, no. 1, pp. 1–8, 1987, ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(87)90037-0.

[223] A. Lodi, S. Martello, and M. Monaci, "Two-dimensional packing problems: A survey," *European Journal of Operational Research*, vol. 141, no. 2, pp. 241–252, 2002, ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(02)00123-6.

[224] M. Dell'Amico, F. Furini, and M. Iori, "A branch-and-price algorithm for the temporal bin packing problem," *Computers & Operations Research*, vol. 114, p. 104 825, 2020, ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2019.104825.

[225] J. Martinovic, N. Strasdat, and M. Selch, "Compact integer linear programming formulations for the temporal bin packing problem with fire-ups," *Computers & Operations Research*, vol. 132, p. 105 288, 2021, ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2021.105288.

[226] J. Legriel, C. Le Guernic, S. Cotton, and O. Maler, "Approximating the pareto front of multi-criteria optimization problems," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. Esparza and R. Majumdar, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 69–83, ISBN: 978-3-642-12002-2.

[227] V. Bracke, J. Santos, T. Wauters, F. De Turck, and B. Volckaert, "A multiobjective metaheuristic-based container consolidation model for cloud application performance improvement," *Journal of Network and Systems Management*, vol. 32, no. 3, p. 61, 2024.

[228] Y. Hu, C. De Laat, and Z. Zhao, "Multi-objective container deployment on heterogeneous clusters," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 592–599. DOI: 10.1109/CCGRID.2019.00076.

[229] *Amazon ec2 spot instances*, https://aws.amazon.com/ec2/spot/instance-advisor/?nc1=h_ls, [Online; accessed 16-September-2025].

[230] J. Xu *et al.*, "Prediction-guided multi-objective reinforcement learning for continuous robot control," in *International conference on machine learning*, PMLR, 2020, pp. 10 607–10 616.

[231] A. Rasheed, O. San, and T. Kvamsdal, "Digital twin: Values, challenges and enablers from a modeling perspective," *IEEE Access*, vol. 8, pp. 21 980–22 012, 2020. DOI: 10.1109/ACCESS.2020.2970143.

[232] M. Fogli, C. Giannelli, F. Poltronieri, C. Stefanelli, and M. Tortonesi, "Chaos engineering for resilience assessment of digital twins," *IEEE Transactions on Industrial Informatics*, pp. 1–9, 2023. DOI: 10.1109/TII.2023.3264101.

[233] S. Tuli, G. Casale, and N. R. Jennings, "Dragon: Decentralized fault tolerance in edge federations," *IEEE Transactions on Network and Service Management*, vol. 20, no. 1, pp. 276–291, 2023. DOI: 10.1109/TNSM.2022.3199886.

[234] D. Borsatti, W. Cerroni, L. Foschini, *et al.*, "Kubetwin: A digital twin framework for kubernetes deployments at scale," *IEEE Transactions on Network and Service Management*, vol. 21, no. 4, pp. 3889–3903, 2024. DOI: 10.1109/TNSM.2024.3405175.

[235] E. Jafarnejad Ghomi, A. M. Rahmani, and N. N. Qader, "Applying queue theory for modeling of cloud computing: A systematic review," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 17, e5186, 2019. DOI: https://doi.org/10.1002/cpe.5186.

[236] *Kubernetes: Scheduling framework*, https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/, [Online; accessed 23-September-2025].

[237] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surv.*, 2022, ISSN: 0360-0300. DOI: 10.1145/3539606.

[238] *Kubernetes: Horizontal pod autoscaling*, https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, [Online; accessed on 23-September-2025].

[239] M. Mendula, A. Bujari, L. Foschini, and P. Bellavista, "A data-driven digital twin for urban activity monitoring," in *IEEE Symposium on Computers and Communications (ISCC)*, 2022, pp. 1–6. DOI: 10.1109/ISCC55528.2022.9912914.

[240] G. Nardini and G. Stea, "Using network simulators as digital twins of 5G/B5G mobile networks," in *IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2022, pp. 584–589. DOI: 10.1109/WoWMoM54355.2022.00091.

[241] D. Van Huynh, V.-D. Nguyen, V. Sharma, O. A. Dobre, and T. Q. Duong, "Digital twin empowered ultra-reliable and low-latency communications-based edge networks in industrial iot environment," in *ICC - IEEE International Conference on Communications*, 2022, pp. 5651–5656. DOI: 10.1109/ICC45855.2022.9838860.

[242] G. Avecilla, J. N. Chuong, F. Li, G. Sherlock, D. Gresham, and Y. Ram, "Neural networks enable efficient and accurate simulation-based inference of evolutionary parameters from adaptation dynamics," *PLoS Biology*, vol. 20, no. 5, e3001633, 2022.

[243] R. Legin, Y. Hezaveh, L. P. Levasseur, and B. Wandelt, "Simulation-based inference of strong gravitational lensing parameters," *arXiv preprint arXiv:2112.05278*, 2021.

[244] M. Sommerfeld and A. Munk, "Inference for empirical wasserstein distances on finite spaces," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 80, no. 1, pp. 219–238, 2018.

[245] C. Bartolini, C. Stefanelli, and M. Tortonesi, "Modeling it support organizations using multiple-priority queues," in *IEEE Network Operations and Management Symposium*, IEEE, 2012, pp. 377–384.

[246] B. Herlicq, A. Khichane, and I. Fajjari, "Nextgenemo: An efficient provisioning of edge-native applications," in *ICC - IEEE International Conference on Communications*, 2022, pp. 1924–1929. DOI: 10.1109/ICC45855.2022.9839012.

[247] Y. Chen, S. Zhang, Y. Jin, Z. Qian, and S. Lu, "Multi-server multi-user game at edges for heterogeneous video analytics," in *ICC - IEEE International Conference on Communications*, 2022, pp. 841–846. DOI: 10.1109/ICC45855.2022.9839250.

[248] C. M. Bishop, "Mixture density networks," English, Aston University, WorkingPaper, 1994.

[249] C. M. Bishop, "Neural networks and their applications," *Review of Scientific Instruments*, vol. 65, pp. 1803–1832, 1994.

[250] M. Allen, D. Poggiali, K. Whitaker, T. Marshall, J. Langen, and R. Kievit, "Raincloud plots: A multi-platform tool for robust data visualization," *Wellcome Open Research*, vol. 4, p. 63, Jan. 2021. DOI: 10.12688/wellcomeopenres.15191.2.

[251] *Keras*, "https://keras.io", [Online; accessed on 24-September-2025].

[252] *Tensorflow: Large-scale machine learning on heterogeneous systems*, "https://www.tensorflow.org/", [Online; accessed on 24-September-2025].

[253] A. Ramdas, N. Garcia, and M. Cuturi, *On Wasserstein Two Sample Testing and related families of nonparametric tests*, 2015. arXiv: 1509.02237 [math.ST]. [Online]. Available: https://arxiv.org/abs/1509.02237.

[254] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2.

[255] J. G. Almaraz-Rivera, "An anomaly-based detection system for monitoring kubernetes infrastructures," *IEEE Latin America Transactions*, vol. 21, no. 3, pp. 457–465, 2023. DOI: 10.1109/TLA.2023.10068850.

[256] P. Almasan, M. Ferriol-Galmés, J. Paillisse, *et al.*, "Network digital twin: Context, enabling technologies, and opportunities," *IEEE Communications Magazine*, vol. 60, no. 11, pp. 22–27, 2022. DOI: 10.1109/MCOM.001.2200012.

[257] J. Zerwas, P. Krämer, R.-M. Ursu, *et al.*, "Kapetánios: Automated kubernetes adaptation through a digital twin," in *2022 13th International Conference on Network of the Future (NoF)*, 2022, pp. 1–3. DOI: 10.1109/NoF55974.2022.9942649.

[258] M. Zaccarini, D. Borsatti, W. Cerroni, *et al.*, "Telka: Twin-enhanced learning for kubernetes applications," in *2024 IEEE Symposium on Computers and Communications (ISCC)*, 2024, pp. 1–6. DOI: 10.1109/ISCC61673.2024.10733736.

[259] A. Blohowiak, A. Basiri, L. Hochstein, and C. Rosenthal, "A platform for automating chaos experiments," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2016, pp. 5–8. DOI: 10.1109/ISSREW.2016.52.

[260] J. Santos, T. Wauters, and F. D. Turck, "Efficient management in fog computing," in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–6. DOI: 10.1109/NOMS56928.2023.10154219.

[261] *Kubetwin, twinning kubernetes applications at scale*, "https://www.kubetwin.org/", [Online; accessed on 20-November-2025].

Author's Publications List

[1] C. Barz, E. Cramer, R. Fronteddu, *et al.*, "Enabling adaptive communications at the tactical edge," in *MILCOM 2022 - 2022 IEEE Military Communications Conference (MILCOM)*, 2022, pp. 1038–1044. DOI: 10.1109/MILCOM55135.2022.10017459.

[2] D. Borsatti, W. Cerroni, L. Foschini, *et al.*, "Modeling digital twins of kubernetes-based applications," in *2023 IEEE Symposium on Computers and Communications (ISCC)*, 2023, pp. 219–224. DOI: 10.1109/ISCC58397.2023.10217853.

[3] L. Manca, D. Borsatti, F. Poltronieri, *et al.*, "Characterization of microservice response time in kubernetes: A mixture density network approach," in *2023 19th International Conference on Network and Service Management (CNSM)*, 2023, pp. 1–9. DOI: 10.23919/CNSM59352.2023.10327842.

[4] R. Galliera, M. Zaccarini, A. Morelli, *et al.*, "Learning to sail dynamic networks: The marlin reinforcement learning framework for congestion control in tactical environments," in *MILCOM 2023 - 2023 IEEE Military Communications Conference (MILCOM)*, 2023, pp. 424–429. DOI: 10.1109/MILCOM58377.2023.10356270.

[5] F. Poltronieri, C. Stefanelli, M. Tortonesi, and M. Zaccarini, "Reinforcement learning vs. computational intelligence: Comparing service management approaches for the cloud continuum," *Future Internet*, vol. 15, no. 11, 2023. DOI: 10.3390/fi15110359.

[6] M. Zaccarini, B. Cantelli, M. Fazio, *et al.*, "Voice: Value-of-information for compute continuum ecosystems," in *2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN)*, 2024, pp. 73–80. DOI: 10.1109/ICIN60470.2024.10494500.

[7] D. Borsatti, W. Cerroni, L. Foschini, *et al.*, "Kubetwin: A digital twin framework for kubernetes deployments at scale," *IEEE Transactions on Network and Service Management*, vol. 21, no. 4, pp. 3889–3903, 2024. DOI: 10.1109/TNSM.2024.3405175.

[8] J. Santos, M. Zaccarini, F. Poltronieri, *et al.*, "Efficient microservice deployment in kubernetes multi-clusters through reinforcement learning," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, 2024, pp. 1–9. DOI: 10.1109/NOMS59830.2024.10575912.

[9] M. Zaccarini, M. Tortonesi, and F. Poltronieri, "The evolution of kubernetes management: Introducing the kubetwin framework," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, 2024, pp. 1–4. DOI: 10.1109/NOMS59830.2024.10575138.

[10] M. Zaccarini, D. Borsatti, W. Cerroni, *et al.*, "Telka: Twin-enhanced learning for kubernetes applications," in *2024 IEEE Symposium on Computers and Communications (ISCC)*, 2024, pp. 1–6. DOI: 10.1109/ISCC61673.2024.10733736.

[11] N. Di Cicco, F. Poltronieri, J. Santos, *et al.*, "Multi-objective scheduling and resource allocation of kubernetes replicas across the compute continuum," in *2024 20th International Conference on Network and Service Management (CNSM)*, 2024, pp. 1–9. DOI: 10.23919/CNSM62983.2024.10814307.

[12] J. Santos, M. Zaccarini, F. Poltronieri, *et al.*, "Hephaestusforge: Optimal microservice deployment across the compute continuum via reinforcement learning," *Future Generation Computer Systems*, vol. 166, 2025. DOI: 10.1016/j.future.2024.107680.

[13] M. Zaccarini, F. Poltronieri, C. Stefanelli, and M. Tortonesi, "Hybridized hot restart via reinforcement learning for microservice orchestration," in *NOMS 2025-2025 IEEE Network Operations and Management Symposium*, 2025, pp. 1–7. DOI: 10.1109/NOMS57970.2025.11073715.

[14] M. Zaccarini, F. Poltronieri, D. Borsatti, *et al.*, "Chaos engineering based kubernetes pod rescheduling through deep sets and reinforcement learning," in *NOMS 2025-2025 IEEE Network Operations and Management Symposium*, 2025, pp. 1–7. DOI: 10.1109/NOMS57970.2025.11073590.

List of Figures

List of Tables

List of Acronyms

A2C	Advantage Actor Critic
AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
BF1B1	Best Fit
BS	Base Station
C2E	Cloud2Edge
CE	Chaos Engineering
CI	Computational Intelligence
CC	Compute Continuum
CCTV	Closed Circuit Television
CNCF	Cloud Native Computing Foundation
DC	Data Center
DNS	Domain Name System
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
DS	Deep Sets
DT	Digital Twin
EC2	Elastic Compute Cloud
ECT	Enforced ConstrainT
FCFS	First-Come First-Served
FFI	First Fit Increasing
FFD	First Fit Decreasing
FIFO	First In First Out
FNN	Feedforward Neural Network
GA	Genetic Algorithm
GMM	Gaussian Mixture Model
GPS	Global Positioning System
GTM	Global Topology Manager
GWO	Grey Wolf Optimization
HPA	Horizontal Pod Autoscaling

IaaS	Infrastructure-as-a-Service
IETF	Internet Engineering Task Force
ILP	Integer Linear Programming
IO	Information Object
IoT	Internet of Things
K8s	Kubernetes
Karmada	Kubernetes Armada
KT	KubeTwin
KubeFed	Kubernetes Federation
MDN	Mixture Density Network
MDP	Markov Decision Process
MEC	Multi-Access Edge Computing
ML	Machine Learning
MO	Multi-Objective
MOO	Multi-Objective Optimization
MLP	Multi-Layer Perceptron
MPSO	Multi-Swarm PSO
MSE	Mean Squared-Error
NFC	Near Field Communication
NFV	Network Function Virtualization
NSGA	Non-Dominated Sorting Genetic Algorithm
PaaS	Platform-as-a-Service
PDF	Probability Density Function
PE	Permutation-Equivariant
PF	Pareto Front
PI	Permutation-Invariant
PPO	Proximal Policy Optimization
PSO	Particle Swarm Optimization
PSR	Percentage of Satisfied Requests
QoS	Quality of Service
QPSO	Quantum Particle Swarm Optimization
RFID	Radio Frequency Identification
RL	Reinforcement Learning
RPS	Requests per Second
S3	Simple Storage Service
SaaS	Software-as-a-Service
SMA	Simple Moving Average
SPF	"Sieve, Process, and Forward"
SNMP	Simple Network Management Protocol
TELKA	Twin-Enhanced Learning for Kubernetes Applications
TRPO	Trust Region Policy Optimization

TTR	Time To Resolution
V2I	Vehicle-to-Infrastructure
V2V	Vehicle-to-Vehicle
VM	Virtual Machine
VoI	Value-of-Information
VOICE	Value-of-Information for Compute Continuum Ecosystems
VPA	Vertical Pod Autoscaling
VPN	Virtual Private Network