Reinforcement Learning-based Rescheduling of Microservice Architecture Applications in the Cloud-Edge Continuum

by Xu Bai

Student Number: 1374321

Supervised by

Dr. Tawfiq Islam, Assoc. Prof. Adel N. Toosi and Prof. Rajkumar Buyya

A thesis submitted in total fulfillment for the degree of Master of Computer Science

in the

School of Computing and Information Systems

Faculty of Engineering and IT

THE UNIVERSITY OF MELBOURNE

October 2024

THE UNIVERSITY OF MELBOURNE

Abstract

School of Computing and Information Systems
Faculty of Engineering and IT

Master of Computer Science

by Xu Bai

The rapid expansion of Internet of Things (IoT) applications has led to increased demand for low-latency processing, a requirement often unmet by traditional centralized cloud computing due to their inherent latency limitations. Edge computing mitigates this issue by placing computational resources closer to end-users, thereby reducing latency; however, it encounters constraints in computational capacity. The cloud-edge continuum architecture addresses these limitations by creating a cloud-edge hybrid environment. Within the cloud-edge continuum, Microservice Architecture (MSA) is increasingly favored, as it decomposes applications into independent, loosely coupled services that can be strategically deployed across both cloud and edge. Yet, the diverse and heterogeneous nature of resources within this environment presents significant challenges for optimal MSA placement to achieve low end-to-end latency. This thesis proposes a novel rescheduling algorithm specifically designed to optimize microservice placement within cloud-edge continuum by adaptively adjusting service placement in real-time. The algorithm employs a reinforcement learning-based approach to develop a rescheduling policy by interacting with the cloud-edge continuum environment, learning the intricate patterns of service invocation and heterogeneous resource availability in hybrid cloud-edge settings. The proposed rescheduling algorithm undergoes extensive evaluation on a real-world testbed. Compared to baseline algorithms, it demonstrates a significant reduction in average end-to-end latency by 7.8%, 11.4%, and 8.8% across three benchmark MSA applications during node failure scenarios. In these scenarios, it also shows impressive results in decreasing latency fluctuations and spikes due to changes in resource availability.

Declaration of Authorship

I, Xu Bai, declare that this thesis titled, "Reinforcement Learning-based Rescheduling of Microservice Architecture Applications in the Cloud-Edge Continuum." and the work presented in it are my own. I confirm that:

- this thesis does not contain any material previously submitted for a degree or diploma at any institution without proper acknowledgment. This thesis, to the best of my knowledge, includes no material previously published or authored by another individual without appropriate citation within the text.
- this thesis did not require clearance from the University's ethics committee.
- this thesis is approximately 28000 words in length, excluding text in figures, tables, code listings, bibliographies, and appendices.

Signed: Xu Bai

Date: 31 October 2024

Preface

The sections 3.1 and 3.2, which review the related works on modeling the Cloud-Edge Continuum and Microservice Architecture (MSA) applications, are adapted from the research proposal completed as part of the requirements for the Master of Computer Science degree.

All content I contributed to this thesis underwent independent review by my supervisors, Dr. Tawfiq Islam, Associate Prof. Adel N. Toosi and Prof. Rajkumar Buyya. Their guidance and feedback, received at various stages during the thesis creation, were incorporated into the final document.

Publications

Title	Author	Venue	Status
Reinforcement Learning-based Rescheduling of Microservice Architecture Applications in the Cloud-Edge Continuum.	Xu Bai, Tawfiq Islam, Adel N. Toosi, Rajkumar Buyya	IEEE Transactions on Parallel and Distributed Systems	In Submission

Acknowledgements

First and foremost, I would like to extend my heartfelt gratitude to my primary supervisor, Dr. Tawfiq Islam, who has guided me throughout my master's journey. From helping me define the research topic to identifying key directions, Dr. Islam has always been approachable and supportive, encouraging me to ask questions freely and answering them with patience and expertise. His detailed reviews and feedback were invaluable in ensuring I completed this thesis before the deadline.

I am also profoundly thankful to my co-supervisor, Assoc. Prof. Adel N. Toosi, for his insightful guidance on my research methodology and experimental approach. He has been exceptionally generous in offering advice, both on research and career planning, which helped clear many uncertainties along the way. This thesis has greatly benefited from his constructive suggestions.

My deepest appreciation also goes to my co-supervisor, Prof. Rajkumar Buyya, who provided critical insights and directions. Through discussions on my research objectives and future planning, he helped me refine my research goals and offered advice that was instrumental in shaping the course of this work.

I feel truly fortunate to have had the support of these three supervisors during my master's degree journey.

Lastly, I am grateful to my family and friends for their unwavering support, which has been my greatest source of strength. A special thanks to my girlfriend, who stood by me through the most challenging times.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing has transformed information technology with its scalable service models, diverse resource offerings, and cost advantages over traditional infrastructures [1]. However, the rapid expansion of Internet of Things (IoT) devices has heightened the demand for low-latency applications, revealing the limitations of centralized cloud data centers, which often struggle to meet strict latency requirements [2]. In response, edge computing has emerged as a complementary paradigm. By positioning computing nodes closer to end-users, edge computing facilitates the rapid processing and analysis of data near its source, significantly reducing latency and enhancing application responsiveness [3]. Despite the low-latency benefits of edge nodes, they typically possess limited computing power compared to cloud data centers. The cloud-edge continuum architecture has been proposed to leverage the advantages of both cloud and edge computing, integrating these resources into a unified framework [4]. As illustrated in Figure 1.1, a typical cloud-edge continuum consists of two layers: the cloud layer, often provided by public platforms such as AWS or Google Cloud, and the edge layer, comprising on-premise devices located near end-users. These layers collaborate to ensure that deployed applications meet Quality of Service (QoS) requirements, balancing the cloud's vast resources with the low-latency benefits of edge devices. Applications deployed in the cloud-edge continuum are increasingly adopting the microservice architecture (MSA) model [5], which has emerged as a popular modular development approach. MSA breaks down software functionalities into independent, loosely coupled services that communicate via network protocols [6]. Coupled with container virtualization technologies like Docker [1] and container orchestration platforms such as Kubernetes [2], MSA enables the flexible placement of these services across heterogeneous cloud and edge resources.

[1]https://www.docker.com/

[2]https://kubernetes.io/

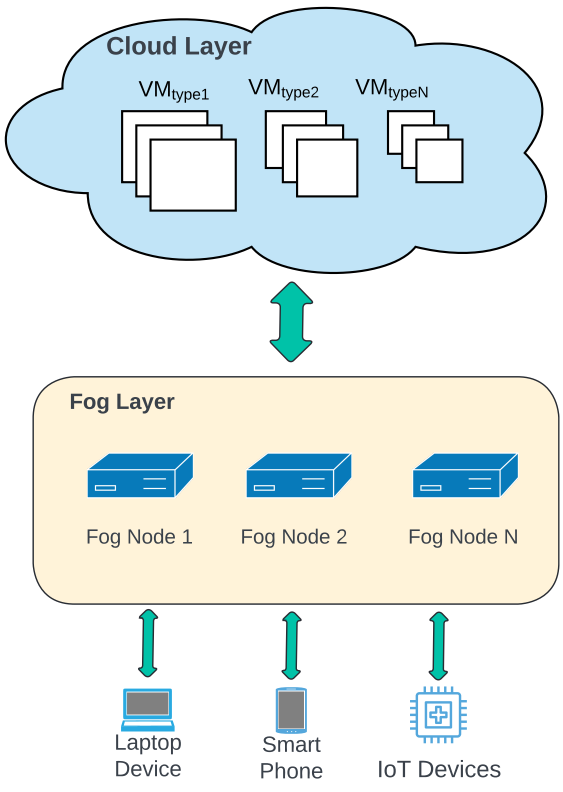


Figure 1.1: Architecture of Cloud-Edge Continuum

Since each service in an MSA application can have distinct computational and latency requirements, they can be strategically placed on different nodes within the cloud-edge continuum to optimize performance [7]. For instance, latency-sensitive services can be deployed on edge nodes to provide fast response times, while compute-intensive tasks can be offloaded to cloud nodes with greater processing capacity. This makes the placement of MSA services critical in effectively harnessing the low-latency advantages of edge resources and the high computational power of cloud resources. Consequently, there is growing research interest in developing efficient MSA placement strategies that can maximize the benefits of both cloud and edge layers in the cloud-edge continuum. The MSA application placement problem in the cloud-edge continuum has proved to be challenging due to both the complicated service invocation patterns and heterogeneous computing resources in the cloud-edge continuum [8].

Existing literature overlooks the complexities of invocation patterns in microservices, which involve intricate dependencies and demand precise orchestration to achieve optimal performance and low latency. Furthermore, the heterogeneous nature of computing resources—from high-capacity cloud servers to resource-limited edge devices—adds additional layers of complexity to the decision-making process for effective microservice deployment. This gap underscores the need for innovative approaches, particularly those utilizing machine learning-based rescheduling techniques. By applying learning algorithms, systems can dynamically adapt to fluctuating workloads and complex service interactions, optimizing resource allocation, enhancing overall system performance, and effectively managing the challenges inherent in the cloud-edge continuum.

1.1 Objectives and Contributions

To tackle the challenges identified, we propose MSA application rescheduling algorithms aimed at optimizing the end-to-end latency of MSA applications. Our approach models the heterogeneous computing resources across the cloud-edge continuum. The end-to-end latency is determined by both network delays and the execution times of all services involved in handling user requests. Our model evaluates these factors by differentiating the execution times of services running on different types of computing resources, while also considering the network latency of edge and cloud layers.

We formulate the placement problem as a rescheduling challenge. Most recent studies like [9–11] treat microservice architecture (MSA) placement as a one-time scheduling task, where services are deployed based on the system's state at a specific moment and then run unchanged until their lifecycle ends. This initial placement lacks ongoing control over the services' lifecycle. However, in the cloud-edge continuum, computing resources are in constant flux: services scale in real-time, new edge devices connect to the network, and cloud nodes are added or removed based on current resource utilization. In such a dynamic environment, one-time placement decisions may fail to maintain consistent service quality throughout the services' lifecycle. On the other hand, rescheduling every service in an MSA application from scratch for every system change is both costly and disruptive. To address this, we propose a rescheduling algorithm that continuously monitors the state of the cloud-edge environment and incrementally reschedules microservices with minimal steps. This approach minimizes application downtime and prevents unnecessary disruptions. We argue that our method is more resilient than traditional placement schemes. Furthermore, the proposed rescheduling algorithm integrates seamlessly with Kubernetes, working in tandem with its autoscaler to optimize microservice placement performance.

Therefore, this thesis addresses to following research questions:

- ***Research Question 1:*** How can we design MSA application rescheduling algorithms that effectively optimize end-to-end latency for delay-sensitive applications deployed in the cloud-edge continuum, considering the challenges posed by heterogeneous resources, intricate MSA network topology, and dynamic changes in resource availability?

- ***Research Question 2:*** How can we utilize reinforcement learning (RL) to implement a dynamic rescheduling mechanism that adapts to real-time changes in the cloud-edge environment, while minimizing rescheduling overhead and ensuring long-term optimal placement of microservices?

In summary, our **key contributions** include:

- We developed a novel MSA rescheduling algorithm that dynamically reschedules microservices, maintaining application performance when cluster availability changes.
- We applied a reinforcement learning-based approach to devise a rescheduling policy mindful of complex microservice topologies and heterogeneous cloud-edge resources.
- We developed a Reinforcement Learning (RL) model with a reward design tailored for minimizing end-to-end latency, aimed at maximizing long-term performance.
- We designed and implemented a custom RL simulation environment that models heterogeneous computing resources and network conditions, reflecting real-world cloud-edge continuum scenarios.
- We integrated the proposed rescheduling algorithm into a real-world Kubernetes testbed, showing substantial improvements over baseline methods in reducing end-to-end latency, minimizing request latency fluctuations, and preventing latency spikes under node failure conditions.

Chapter 2

Background

In this chapter, we begin by outlining the development trajectory of the Cloud-Edge Continuum in Section 2.1. Following this, Section 2.2 introduces the concept of Microservice Architecture (MSA) applications, highlighting their modular and scalable nature. We further examine the MSA application placement problem, a critical aspect that determines how effectively these applications are deployed across cloud and edge resources to optimize performance and resource utilization. Lastly, Section 2.3 provides an overview of container orchestrators, key tools that facilitate the efficient scheduling, scaling, and management of MSA applications in Cloud-Edge Continuum.

2.1 Cloud-Edge Continuum

2.1.1 Cloud Computing

Over the past few decades, the landscape of computing architectures has undergone significant transformations, driven by exponential growth in data generation, the need for scalable processing power, and the demand for low-latency applications. Initially, companies and organizations deployed their applications or services on monolithic computing machines [12]. With advancements in computer technology, the processing capabilities of single machines increased exponentially, as predicted by Moore's Law [13]. The processing power of individual CPUs continuously improved, and the emergence of multi-core processors further enhanced computational capacity. For a time, applications deployed on single machines sufficed for most computational tasks and service demands. However, as computational tasks grew increasingly complex, single-machine processing power became insufficient for handling intricate computational tasks.

In response to these limitations, Grid Computing was proposed [14], leveraging multiple machines with relatively low computing power to perform large-scale data processing and scientific computations. However, Grid Computing is primarily utilized by research communities for complex scientific computing. The emergence of Cloud Computing is changing this landscape [12]. Similar to Grid Computing, Cloud Computing utilizes multiple machines collectively for computational tasks. However, Cloud Computing, unlike Grid Computing, has a broader range of applications from lightweight application service deployment to heavy computational batch jobs.

In Cloud Computing, a vast pool of computational resources is centrally managed by public cloud providers like Google or Amazon through robust cloud management platforms, such as OpenStack [1]. Utilizing virtualization technology, public cloud providers are able to isolate and allocate computing resources to various users. This centralization offers several key advantages: users avoid the financial burden of purchasing and maintaining physical hardware, paying instead only for the computing time they consume, resulting in significant cost savings. Additionally, cloud resources are highly scalable, allowing users to seamlessly adjust resource allocation based on fluctuating workloads [1]. This flexibility enables quicker deployment of applications and services, making Cloud Computing both accessible and adaptable to varying computational demands.

2.1.2 Edge Computing

As Cloud Computing continues to develop, its shortcomings are becoming apparent. For applications requiring real-time processing and low latency, the inherent delay in transmitting data to centralized cloud servers can be problematic. Furthermore, in cases like autonomous vehicles, healthcare IoT, and augmented reality, the data generated is massive, incurring severe bandwidth pressure on the backbone network when transmitting such data to cloud computing nodes [3].

Edge computing has emerged as a complementary paradigm, positioning computing resources closer to end-users to address the latency and bandwidth issues associated with centralized cloud computing. In this model, user applications are primarily hosted on edge nodes, leveraging their proximity to reduce latency and enhance response times. By routing all network traffic directly to these edge nodes, edge computing minimizes latency and alleviates potential bandwidth congestion between cloud resources and end-users. Here, cloud nodes function primarily as control units, overseeing the management of edge nodes and gathering metrics and analytical data rather than directly hosting applications.

[1]https://www.openstack.org/

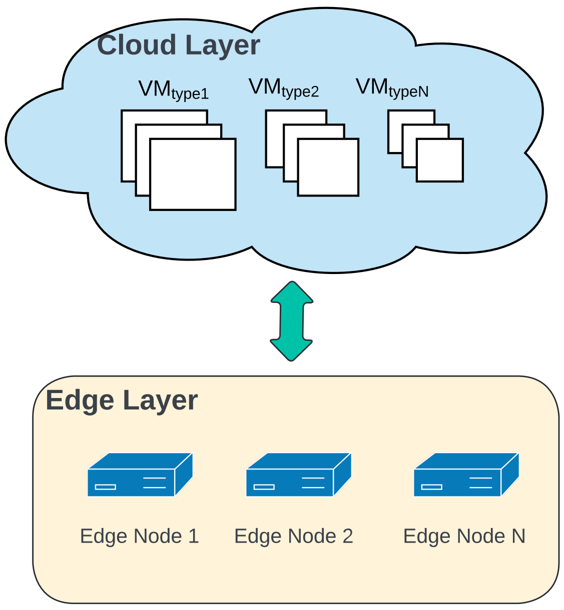


FIGURE 2.1: Cloud-Edge Continuum Architecture

2.1.3 Cloud-Edge Continuum

Even though edge computing offers low-latency service hosting close to end-users, its computing resources are often constrained. In self-hosted scenarios, for instance, edge nodes possess significantly lower computational power compared to cloud nodes, limiting their ability to manage computationally intensive tasks effectively [4]. To capitalize on the strengths of both cloud and edge computing, the Cloud-Edge Continuum has emerged as an integrated architecture that leverages the capabilities of both to achieve optimal performance. As illustrated in Figure 2.1, this architecture comprises two principal layers: the cloud layer, which hosts cloud computing resources, and the edge layer, which accommodates all edge resources. Unlike traditional edge computing, where cloud nodes primarily function as control nodes without directly hosting user applications, the Cloud-Edge Continuum allows cloud nodes to also serve as computing nodes that can execute user applications [4].

By seamlessly distributing workloads between centralized cloud data centers and decentralized edge nodes, the continuum addresses the diverse requirements of modern applications. However, the placement of applications in the Cloud-Edge Continuum raises challenges. Due to the heterogeneous computing resources in the cloud and edge layers, the placement of applications onto specific nodes in the continuum can significantly impact end-user latency, costs, and network usage [7]. In this work, we will delve into the application placement problem, more specifically, the Microservice Architecture Application (MSA) placement problem in the cloud-edge continuum.

2.2 Microservice Architecture Application

As the software industry continues to evolve, software architectures are also undergoing significant development. In the early stages of the software industry, most applications were deployed as monoliths, meaning that all functional components and business logic were contained within a single, tightly integrated system. However, this architecture often led to high coupling between internal components, resulting in difficulties in software maintenance and significant overhead when developing new features compatible with existing ones [15]. Moreover, monolithic applications typically cannot scale computing resources for specific modules; scaling must occur at the application level, which reduces flexibility and scalability.

To address the shortcomings of monolithic applications, Microservice Architecture (MSA) has gained substantial adoption [16]. Compared to traditional monolithic applications, an MSA application decomposes each module into a collection of loosely coupled services, each responsible for a specific functionality such as user authentication, data processing, or web front-end services. These services are independent, enabling teams to develop, test, and deploy each service in isolation, often using different technologies or programming languages based on the needs of the service [17].

2.2.1 Scalability and Fault Tolerance

Figure 2.2 illustrates an example of a healthcare Microservice Architecture (MSA) application used for gathering and analyzing patient data [18], often deployed within the Cloud-Edge Continuum. In this example, the application's functionality is distributed across three intercommunicating microservices. The patient's vital sign data, generated by IoT devices, is first sent to a data compression service, which reduces the data size for more efficient transmission and storage. The compressed data is then forwarded to two distinct services: a machine learning service for long-term analytics and an emergency service for real-time anomaly detection in patient conditions. Each of the services in this MSA application has two replicated instances for load balancing and fault tolerance purposes.

As demonstrated in the example MSA application, each microservice can scale to multiple instances to adapt to fluctuating workloads. Moreover, they can be distributed across different computing nodes to enhance the application's fault tolerance. The advantages of utilizing multiple replicas are twofold. First, it significantly enhances the system's

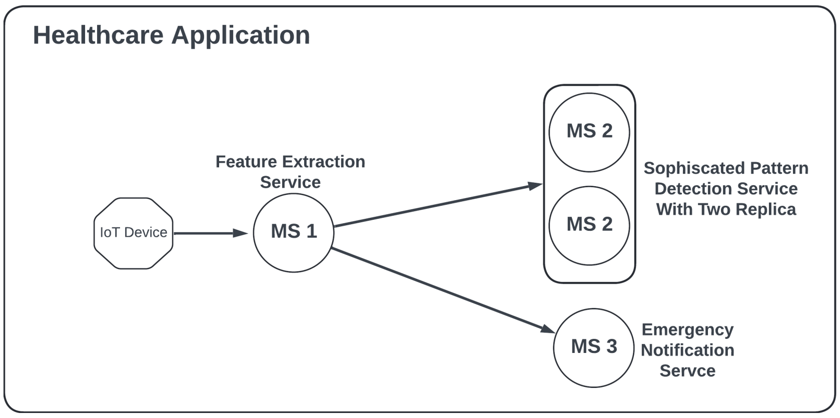

Healthcare Application
IoT Device
Feature Extraction Service
MS 1
MS 2
MS 2
Sophiscated Pattern Detection Service With Two Replica
MS 3
Emergency Notification Servce

FIGURE 2.2: MSA Application Example

capacity to handle varying levels of throughput. In modern software deployment platforms like Kubernetes [2], MSA applications can leverage auto-scaling mechanisms [19] that dynamically adjust the number of service replicas based on demand. For instance, when IoT data in the example healthcare application surges, the platform automatically scales up by adding replicas to accommodate the increased load. Conversely, during periods of low demand, the number of replicas is reduced to conserve resources, thereby optimizing system efficiency.

Second, employing multiple replicas reduces the risk of a single point of failure [19]. If one replica of a service fails, others can continue to operate without disrupting the overall application, ensuring high availability and reliability. In summary, the modular nature of MSA not only provides flexibility in adapting to dynamic workloads but also enhances resilience to failures, making it a highly robust architecture for modern distributed systems.

2.2.2 Network Topology

Although the modularity of Microservice Architecture provides significant flexibility and availability, it also introduces substantial complexity, particularly in terms of increased network invocations between services [15]. This presents a critical challenge in optimizing MSA application performance. Services in an MSA application communicate with each other using network protocols such as HTTP [3], WebSocket [4], or gRPC[5] to collaboratively process a user request. The invocation patterns in MSA applications are highly diverse. Figure 2.3 illustrates three common MSA network topologies: the "chain" pattern, where one service invokes another sequentially—often seen in batch

[2]https://kubernetes.io/
[3]https://datatracker.ietf.org/doc/html/rfc2616
[4]https://datatracker.ietf.org/doc/html/rfc6455
[5]https://grpc.io/

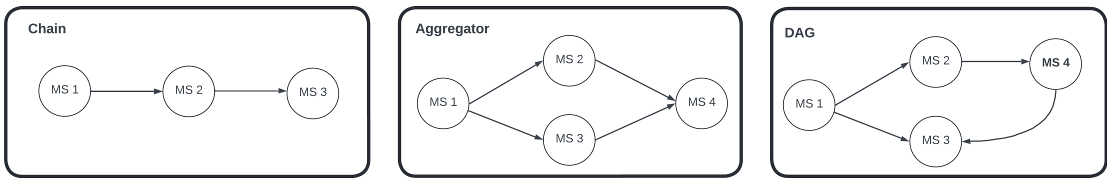


FIGURE 2.3: Invocation Patterns of MSA Application

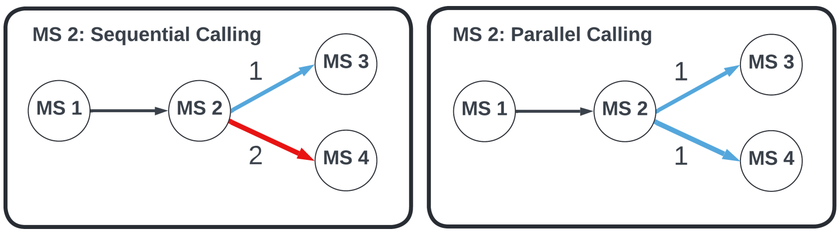


FIGURE 2.4: Microservice With Different Invocation Order

processing tasks; the "aggregator" pattern, where a service calls multiple other services and aggregates the results; and the "Directed Acyclic Graph (DAG)" pattern, which involves a more complex structure of services calling multiple external services to deliver a response to the end user. In real-world workloads, the network topologies of MSA applications can be far more intricate, often composed of different invocation patterns. Another layer of complexity in MSA application arises from the order in which services are invoked. Most modern MSA applications employ multi-processing or multi-threading mechanisms to invoke services asynchronously, thereby increasing throughput [15]. Services that do not depend on each other can be invoked in parallel, while those with dependencies must be invoked sequentially. Figure 2.4 shows two MSA applications with aggregator pattern, where the service invocation order of the aggregator service differs. In MSA application 1, Service A invokes two external services in parallel. In contrast, in MSA application 2, the invocation of Service F is dependent on the result of Service E, requiring sequential invocation. The external latency of Service A in both applications can be calculated as:

$$\text{External Service Invoking Time}_{\text{Service A}} = \max(\text{Latency}_{\text{Service B}}, \text{Latency}_{\text{Service C}}) \quad (2.1)$$

$$\text{External Service Invoking Time}_{\text{Service D}} = \text{Latency}_{\text{Service E}} + \text{Latency}_{\text{Service F}} \quad (2.2)$$

This formula illustrates that in the parallel invocation case, the overall latency is determined by the slower of the two parallel services, while in the sequential case, the latencies accumulate additively. The complexity of service invocation patterns and their execution order presents significant challenges when placing MSA application services, especially in scenarios with stringent latency requirements. This challenge has motivated the research community to explore strategies that minimize end-to-end latency while preserving the modular advantages of MSA.

2.2.3 Microservice Application Placement Problem in Cloud-Edge Continuum

The modularity of MSA applications makes them particularly well-suited for deployment in cloud-edge hybrid environments. Each service in an MSA application can leverage the heterogeneous characteristics of cloud and edge computing resources to optimize overall application performance. However, as mentioned above, the placement of microservices poses significant challenges due to the complex internal service invocation patterns. This complexity is further amplified when deploying MSA applications within the cloud-edge continuum, as the hybrid environment introduces heterogeneous computing resources with varying capabilities and network conditions [10].

Depending on the optimization objectives—such as minimizing latency, maximizing resource utilization, or ensuring high availability—the MSA application placement strategy must consider several factors: the availability and capacity of heterogeneous computing resources across cloud and edge nodes, the varying computational power of each node, and the different latencies between cloud and edge nodes relative to end-users. Additionally, the internal network topologies of MSA applications play a critical role in determining optimal placement.

In this work, we capture the network topology of MSA applications by modeling both the internal service invocation patterns and the external invocation orders—specifically, services invoked in parallel or sequentially. Additionally, we model the heterogeneous cloud and edge computing resources, considering critical factors such as network latency, computational capacity, and resource availability, which will be thoroughly detailed in Sections 4.1.

2.3 Container Orchestrator

2.3.1 Overview

Historically, deploying software across a cluster of heterogeneous computing nodes was a challenging and resource-intensive task [12]. Computing nodes within a cluster could run different operating systems, often requiring DevOps teams to manage complex deployment scripts and tools to ensure software compatibility across various OS versions and configurations. As virtualization machine technology emerged, A cluster of heterogeneous resources can run virtual machines with consistent operating systems, however, traditional virtual machine techniques often incur high overhead due to simulating the underlying hardware of the virtual machines.

The emergence of container virtualization technology has transformed the landscape of managing heterogeneous computing resources. A ”container” refers to an isolated environment used to run software, where all necessary dependencies, runtimes, and files are packaged together. This approach allows applications to run on host machines without installing the software's dependencies, provided the host supports containerization technology. Furthermore, container technology involves resource isolation without hardware emulation, resulting in minimal overhead compared to running virtual machines. Consequently, container virtualization enables software to be easily deployed across heterogeneous computing nodes. To achieve unified management of container applications and clusters, several container orchestrators have been introduced. Following the release of container technology, Docker launched its container orchestrator Docker Swarm [6]. Mesos [7], a widely adopted distributed systems framework, also released its container orchestrator Marathon [8], which manages containers running over Mesos clusters. Among these, the most widely adopted container orchestrator is Kubernetes [9], originally initiated and developed by Google before becoming open source, making it the most active container orchestrator project. Kubernetes natively supports autoscaling, load balancing, and self-healing mechanisms for deployed containers, and its modular design ensures great extensibility and customization, establishing it as the de facto container orchestrator in the industry.

2.3.2 Kubernetes

Given these characteristics, Kubernetes has been widely adopted to orchestrate Microservice Architecture (MSA) applications within the cloud-edge continuum [7]. In a Kubernetes-orchestrated cloud-edge environment, both cloud and edge nodes—despite their diverse characteristics—are collectively managed as a unified cluster. Each microservice in an MSA application can be scheduled to any computing node with sufficient available resources, whether in the cloud or at the edge. In addition to application scheduling, Kubernetes provides features like autoscaling, self-healing, and load balancing, ensuring the scalability, resilience, and availability of applications.

Figure 2.5 illustrates the architecture of Kubernetes. Overall, Kubernetes comprises two types of nodes: the control plane node and the worker nodes. The control plane node maintains the cluster's state, receives API calls from other nodes or end-users, and performs application scheduling along with other control mechanisms. Worker nodes primarily host pods, the smallest deployable units in Kubernetes that encapsulate one

[6]https://docs.docker.com/engine/swarm/
[7]https://mesos.apache.org/
[8]https://mesosphere.github.io/marathon/
[9]https://kubernetes.io/

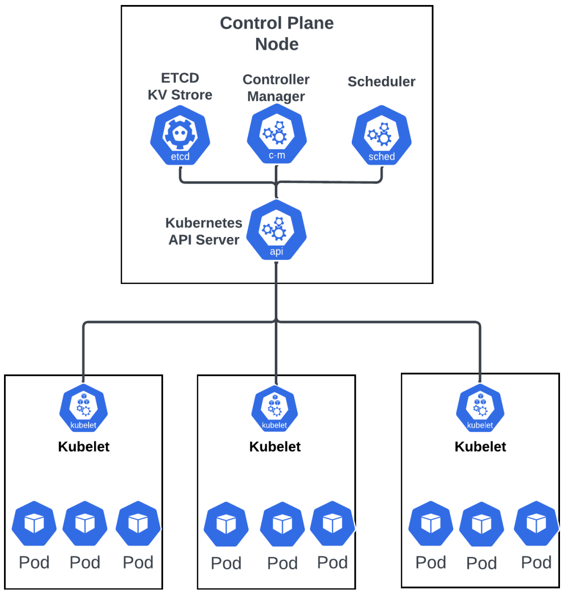


FIGURE 2.5: Kubernetes Architecture

or more containers. Within this client-server framework, several key components form the architecture of Kubernetes:

API Server: The control plane that manages communication between all Kubernetes components and responds to requests from the cluster manager.

Controller Manager: Manages various controllers running in the Kubernetes cluster. The controllers monitor the state of the cluster and resources, ensuring they maintain the desired state.

Scheduler: Allocates application workloads to appropriate nodes based on resource availability and user-defined policies. It plays a key role in application placement.

etcd: A distributed key-value store that stores essential cluster data and, in some cases, application data, including configuration, state, and metadata.

Kubelet: A service running on each Kubernetes-managed node, responsible for ensuring that containers are running as specified in the desired configuration.

The controller manager, scheduler, and API server are all deployed on the control plane node, serving as the core orchestration components responsible for managing the entire cluster. In contrast, the Kubelet is deployed on every worker node, ensuring that the containers running on those nodes adhere to the specified configuration and continue to operate as expected.

2.3.3 Microservice Architecture Application Placement in Kubernetes

When deploying an MSA application in Kubernetes, a set of resources defined in YAML files is used to specify the application's deployment configuration. The smallest deployable computing unit in Kubernetes is the **Pod** resource. A Pod itself is a container that includes user-running code. Users can configure the required CPU and memory resources for the pods and also set limits on pod resource usage. While Pods can be individually deployed within a Kubernetes cluster, a **Deployment** resource can be defined to manage multiple replicated Pods, enabling load balancing and fault tolerance. Furthermore, a **Service** resource can be defined to expose the Pods' endpoints to other services or external users. Any traffic directed to service-defined endpoints will be load-balanced across the associated Pods. Pod, Deployment, and Service resources in Kubernetes declaratively define the deployment of an MSA application, which directly determines the placement of each MSA application component.

As discussed in Section 2.2.3, in the hybrid cloud/edge environment, the placement of each microservice in an MSA application is crucial to ensure optimal application performance. In Kubernetes, the **Scheduler** and **Autoscaler** jointly decide pod placement on specific nodes.

Scheduler: The primary component responsible for MSA placement in the Kubernetes cluster. Based on the underlying Deployment resource specified by users, the scheduler follows a two-phase decision process to determine the appropriate target node for each Pod in the Deployment. The first phase is the **filtering** process, during which computing nodes that cannot meet the Pod's CPU or memory requirements are filtered out. The second phase is the **scoring** process, where the scheduler assigns a score to each candidate node based on a set of default Kubernetes rules and any user-defined custom rules. The node with the highest score is selected as the target for the Pod deployment. This two-step decision process ensures that Pods are placed on nodes with sufficient resources while allowing for customizable placement policies tailored to specific workload needs.

Autoscaler: In Kubernetes, the autoscaler dynamically adjusts the number of replica pods in a Deployment based on metrics such as CPU or memory utilization, as well as other customizable indicators. The autoscaler is responsible for making scaling decisions and invoking the scheduler to place the pods accordingly. Figure 2.6 illustrates the abstract autoscaling process carried out collaboratively by the autoscaler and scheduler. The autoscaler continuously monitors the CPU utilization of the running pod for microservice MS1. When user requests increase, leading to a spike in pod CPU utilization, the autoscaler initiates scaling and triggers the scheduler to deploy additional pods

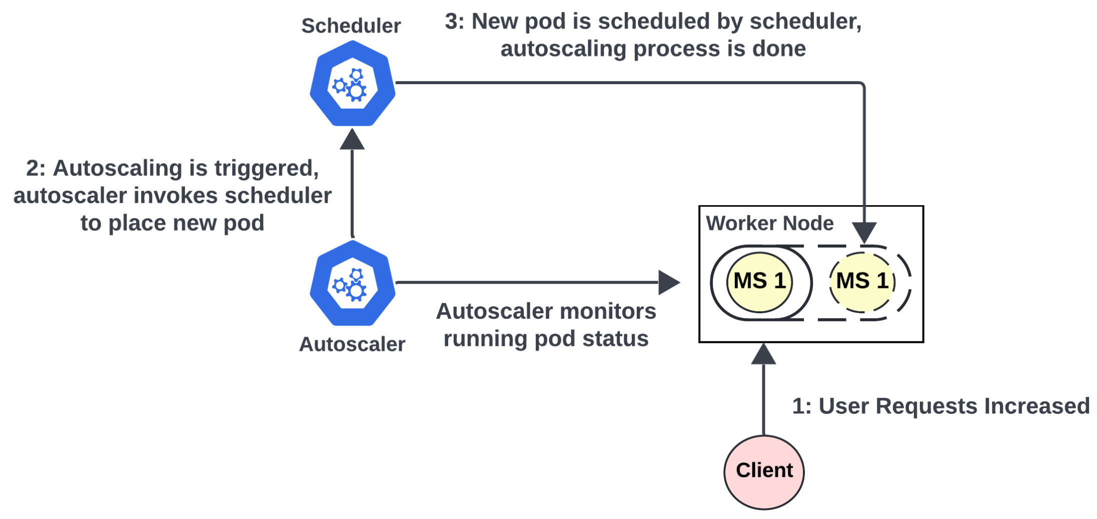

Scheduler
3: New pod is scheduled by scheduler, autoscaling process is done
2: Autoscaling is triggered, autoscaler invokes scheduler to place new pod
Worker Node
MS 1
MS 1
Autoscaler monitors running pod status
Autoscaler
1: User Requests Increased
Client

FIGURE 2.6: Microservice Placement by Autoscaler and Scheduler

within the cluster. The scheduler then allocates a new pod for MS1, thereby completing autoscaling and ensuring that sufficient resources are available to handle the increased workload.

Most existing studies [7, 9, 20, 21] focus on optimizing Microservice Architecture (MSA) application placement during initial scheduling or through autoscaling processes. A key observation regarding Kubernetes pod placement is that the scheduler's decision is one-off, which means that once a pod is placed on a node, it remains there until termination. However, in the Cloud-Edge Continuum, resource availability can change dynamically over time, potentially rendering an initially optimal microservice placement suboptimal as conditions evolve. To address this issue, we propose a rescheduling algorithm that extends the Kubernetes scheduler by enabling dynamic rescheduling of pods at runtime, thereby continuously optimizing MSA application placement in response to changing resource availability.

Chapter 3

Related Works

In this chapter, we begin by reviewing existing research on the modeling of cloud-edge continuum environments in Section 3.1. In Section 3.2, we discuss the challenges and complexities associated with the MSA placement problem in the cloud-edge continuum. Subsequently, we examine the current state-of-the-art scheduling strategies employed to address the MSA placement problem within the cloud-edge continuum in Section 3.3.

3.1 Modeling Cloud-Edge Continuum Environments

Modeling in the Cloud-Edge Continuum introduces greater complexity than in traditional cloud or edge computing environments, which typically includes a wide variety of heterogeneous computing resources, differing in aspects such as computational power and resource availability [5]. Additionally, the continuum spans devices distributed across various locations, complicating network latency modeling and computing resources' spatial proximity within the cloud-edge continuum [22]. This section will review existing studies on modeling heterogeneous computing resources and network conditions in the Cloud-Edge Continuum.

3.1.1 Compute Resource Modeling

Computing resources within the cloud-edge continuum can be modeled at various levels of abstraction, ranging from the physical server infrastructure to virtual machines provided by cloud providers, or even down to individual application containers. The latter aligns with the pay-as-you-go paradigm, where the underlying infrastructure supporting containers is abstracted away from the user. Previous studies, such as [9, 23–25], have focused on modeling physical servers within the edge layer to develop effective service placement strategies. However, these approaches often emphasize placement decisions solely within the edge layer, treating cloud layers primarily as a control plane for scheduling, rather than as active hosts for application components. This limited scope can overlook the potential of cloud resources in dynamic, multi-layered service placement strategies.

Conversely, research like [26–28] suggests that the cloud layer possesses virtually unlimited computing resources, enabling edge nodes to offload computing tasks to the cloud. This approach is designed to compensate for the edge's relatively limited computational capabilities. In these models, edge servers are characterized as nodes with constrained resources but benefiting from low latency, whereas cloud servers are depicted as having boundless computing resources but with higher latency. Such a model is particularly well-suited for integrating edge nodes with cloud-provided services like Container as a Service (CaaS) [29] or Serverless computing [30]. These services typically abstract the underlying cloud infrastructure from developers, allowing the cloud layer to be perceived as an infinitely scalable, pay-as-you-go computing resource. This abstraction simplifies the deployment process and enhances scalability, making it a viable option for expanding the capabilities of edge computing environments.

In studies such as [7, 31, 32], the Cloud-Edge Continuum environment is modeled as an architecture that integrates self-hosted edge nodes with cloud nodes provided by cloud service providers. The edge layer is conceptualized as a cluster of nodes specifically engineered to host microservices with stringent low-latency requirements. In contrast, the cloud layer is envisioned as another cluster of computing nodes with greater resource availability and computing power. Within this model, computing nodes from both cloud and edge environments are viewed as resources available for hosting applications, differing primarily in their computing capacity, resource availability, and network conditions. By explicitly modeling the cloud and edge nodes in this manner, researchers can more precisely optimize application performance based on the heterogeneity of computing nodes across the cloud and edge layers. Therefore, in this work, we adopt this modeling approach of the Cloud-Edge Continuum and design our scheduling algorithms to optimize application placement on both cloud and edge nodes.

3.1.2 Network Modeling

The edge-cloud continuum features a highly complex network topology, where computing nodes in both the edge and cloud layers are geographically dispersed and interconnected through diverse network architectures [5]. The specific network conditions within this continuum can significantly influence the Quality of Service (QoS) of deployed applications. This impact is particularly pronounced for applications based on microservice architectures (MSA), which typically involve numerous internal network interactions. In such applications, the network conditions of the computing nodes hosting each service can substantially affect the overall application performance, especially concerning end-to-end latency. Therefore, modeling the network environment in the edge-cloud continuum is crucial for scheduling algorithms aiming to optimize the performance of MSA-based applications.

Existing research on the MSA application placement problem addresses Cloud-Edge Continuum networking factors with varying degrees of detail. Some studies, such as [33, 34], have not explicitly defined or considered the impact of networking factors, potentially overlooking how network calls between services could influence both the network environment and the services themselves. In contrast, works like [9, 35] have included models of network connectivity between computing nodes, which is a step towards understanding how computing nodes are organized within the network topology.

Furthermore, studies such as [7, 36] have considered the bandwidth of network links, an important factor in network performance. By accounting for bandwidth usage between edge nodes and the backbone network of the edge and cloud layers, application placement strategies can schedule data-intensive applications closer to the edge. This approach can mitigate the load on the backbone network, enhancing overall network efficiency.

Another important network parameter is the distance and latency of computing nodes in the cloud and edge layers, which has been modeled in studies like [25, 28]. A key characteristic of edge nodes in the cloud-edge continuum is their close proximity to end users. Therefore, accounting for the different network latencies between cloud and edge nodes is an important factor when optimizing the end-to-end latency of applications deployed in the cloud-edge continuum. In this work, we model the cloud-edge continuum network by differentiating the computing nodes' latency to the end users, which facilitates our rescheduling algorithms to optimize the end-to-end latency of the underlying MSA applications.

3.2 Microservice Placement Problem

Due to the modular design of Microservice Architecture (MSA) applications, individual services can be independently hosted on different computing nodes within the cloud-edge continuum. However, the placement of each microservice across heterogeneous cloud and edge nodes can significantly influence the overall performance of the MSA application.

Studies such as [9, 32, 37] have developed service placement policies tailored for individual services, achieving improvements in latency, resource utilization, and energy consumption. Nevertheless, this research primarily focuses on optimizing the performance of individual services, thereby overlooking the complex internal network interactions inherent in MSA applications. In this section, we review current research addressing the placement problem of MSA applications. We examine how existing studies address the complicated microservice invocation patterns and define their optimization objectives to enhance overall MSA application performance.

3.2.1 Microservice Invocation Patterns

One of the primary challenges in addressing the placement of MSA applications within the cloud-edge continuum is accounting for the complex invocation patterns and data flows among microservices. Research such as [23, 33, 36] has modeled microservices using a chained structure to depict service interactions within MSA applications, characterizing data flow between services as linear. While this approach simplifies the representation of service dependencies, it may fail to accurately capture the intricate interactions present in real-world applications. In contrast, studies like [7, 9, 26] employ Directed Acyclic Graphs (DAGs) to provide a more detailed representation of data flows within MSA architectures, accommodating non-linear service interactions. In such invocation patterns, microservices can invoke multiple external microservices to execute service tasks, thereby better reflecting the complex communication patterns inherent in modern microservice-based applications.

3.2.2 Microservice Application Placement Objectives

To address the placement problem of MSA application, several objectives can be defined, which can be varied depending on the perspective taken in the research. Previous works, such as [27, 38], prioritize enhancing resource efficiency for edge-cloud continuum service providers, aiming to accommodate more application deployments. Similarly, studies like [32, 39] emphasize energy consumption and cost reduction as key objectives in their placement strategies. Additionally, research such as [35, 40] highlights bandwidth optimization, particularly for data-intensive applications. By positioning these applications closer to the edge, bandwidth requirements for transmitting data to remote cloud layers can be significantly reduced.

Minimizing end-to-end application latency is one of the most critical placement objectives, particularly for applications in the cloud-edge continuum with strict latency requirements [41]. Unlike traditional monolithic applications, where a user request is processed by a single service, MSA applications involve passing the request through multiple microservices. This complexity necessitates accounting for both the execution time of individual services and the network latency when requests are transmitted between them.

Research such as [23, 24] addresses this challenge by modeling MSA applications with chain invocation patterns. In this model, the end-to-end application latency comprises the sum of each service's execution time and its internal network latency. However, in practice, microservices often interact with multiple external services to handle a single request rather than following a simple chain pattern. Therefore, studies like [28, 40] model microservice application invocations as Directed Acyclic Graphs (DAGs). By representing data invocations as DAGs, these studies account for the overall end-to-end latency contributed by different data paths in a single request invocation, which more accurately aligns with real-world MSA application scenarios.

Our research focuses on applications in the cloud-edge continuum with stringent latency requirements, making the reduction of the end-to-end application latency our primary objective. The proposed rescheduling algorithms model MSA applications using DAGs and take into account multiple data paths to optimize the overall MSA application's end-to-end latency.

3.3 Microservice Scheduling Algorithms

3.3.1 Heuristics and Optimization Approaches

To address the MSA placement problem, many studies have utilized heuristics like computing resource availability, application latency, and network bandwidth usage to generate MSA application placement plans Filip et al. [23] propose a heuristic-based scheduling algorithm for microservices, where scheduling decisions are driven by the expected service execution time as a heuristic. This approach aims to optimize service execution within the constraints of heterogeneous computing resources. However, this model simplifies the invocation patterns of MSA applications by assuming a linear chain structure, which does not capture the complexities often present in real-world microservice applications. Moreover, their evaluation is limited to a simulated CloudSim environment, which lacks the variability and intricacies of real-world cloud-edge continuum systems.

Guerrero et al. [28] adopt a heuristic that prioritizes the shortest path from the cloud to computing nodes, rescheduling frequently invoking microservice closer to end-users along this path to minimize latency. Although effective in reducing latency for individual services, this approach fails to account for complex multi-service interactions within MSA applications, resulting in potentially suboptimal placement decisions. Furthermore, their study is confined to a simulation, limiting its applicability to operational cloud-edge environments.

Centofanti et al. [20] address the latency optimization challenge in MSA applications through a latency-aware Kubernetes rescheduling process in a real-world Kubernetes cluster. By leveraging a custom scheduler and de-scheduler, this method iteratively reschedules microservice replicas to nodes with lower end-to-end latency. While effective, this approach optimizes only single-service latency, focusing narrowly on end-to-end latency reduction without accounting for the broader invocation patterns of MSA applications. Consequently, it risks making suboptimal placement decisions that may overlook holistic application requirements.

In addition to heuristic approaches, optimization-based methods are frequently employed to address MSA application placement challenges, often leveraging techniques such as Mixed Integer Linear Programming (MILP) and Particle Swarm Optimization (PSO) to address microservice placement problems with multiple objectives.

Herrera et al. [24] propose an optimization-based approach to address microservice placement in edge computing environments. This approach utilizes MILP to formalize and solve the combined placement of both computing nodes and microservices. However, this approach solely models MSA applications with chain invocation patterns, which is not able to capture more intricate microservice invocation patterns. Furthermore, its validation within a simulated environment limits its applicability to complex, real-world deployment scenarios.

Ying Xie et al. [21] introduce an enhanced Particle Swarm Optimization (PSO) algorithm for MSA scheduling in cloud-edge contexts. PSO is a population-based optimization technique inspired by natural swarm behavior [42]. This work adopts it to address multi-objective requirements such as minimizing both cost and latency for MSA deployments. By dynamically allocating microservices between cloud and edge resources, this approach optimizes for both objectives. However, the lack of real-world deployment evaluation raises questions about its efficiency in practical cloud-edge environments.

Alelyani et al. [43] present a scheduling strategy focused on reducing application latency by scheduling MSA service replicas close to their dependent services to minimize network overhead. This method employs a modified PSO to optimize for latency reduction, primarily by emphasizing the proximity of service replicas. However, by focusing only on proximity within isolated service pairs, this approach neglects the full scope of inter-dependencies between microservices, potentially resulting in suboptimal placement configurations that do not achieve the lowest overall latency for the application.

3.3.2 Reinforcement Learning (RL) Approaches

Due to the complex and dynamic nature of microservice architecture (MSA) application placement, reinforcement learning (RL) approaches have been increasingly applied to address this challenge. Mampage et al. [33] propose a deep reinforcement learning (DRL) model for container scheduling in multi-tenant, resource-constrained serverless computing environments. Utilizing the Deep Q-Network (DQN) approach, they aim to optimize application response time and provider cost efficiency by dynamically scheduling containers to virtual machines (VMs) based on real-time resource demands. However, since their work focuses on individual container scheduling, the interdependencies among different containers are not addressed.

Jayanetti et al. [9] employ a proximal policy optimization (PPO) reinforcement learning model to tackle the service scheduling problem in the cloud-edge continuum. Their approach accounts for both energy efficiency and time optimization. They utilize a hierarchical action space that first determines the layer of the cloud-edge continuum to select and then decides the specific nodes for service scheduling. This method enhances the convergence efficiency of the PPO agent. Nonetheless, their work concentrates on single-service scheduling, rendering it incapable of addressing the placement of MSA applications with intricate interdependencies among multiple services.

Lv et al. [38] introduce an RL-based microservice scheduling algorithm in edge computing, focusing on minimizing communication overhead and balancing load across nodes. The proposed Reward Sharing Deep Q-Learning (RSDQL) enhances the efficiency of the RL agent's learning process. They model the MSA application as a directed acyclic graph (DAG) to capture intricate request invocations. In their subsequent work [34], Lv et al. further employ DRL as the scheduling algorithm to minimize deployment overhead while maintaining the MSA application's latency Quality of Service (QoS) constraints. Additionally, they use a Graph Convolutional Network (GCN) to encode the call graphs of MSA applications, improving the state representation quality for the reinforcement learning agent. However, both works do not model heterogeneous computing resources which limits them to being deployed in a cloud-edge continuum comprised of nodes with varying computational capabilities.

Maia and Ghamri-Doudane [44] propose an RL scheduling approach for scalable microservices across the edge-to-cloud continuum. Their method addresses the multi-objective problem of minimizing deployment costs and reducing application latency.

The RL model iteratively schedules each service and employs heuristic methods for load distribution. However, their work does not model heterogeneous computing resources with varying computational power, limiting its applicability in the cloud-edge continuum. Furthermore, their experiments are conducted solely in simulation environments, which may not fully capture the complexities of real-world deployments.

Chen et al. [45] present a Multiple Buffer Deep Deterministic Policy Gradient (MB_DDPG) RL algorithm designed for microservice deployment in edge-cloud hybrid environments. This algorithm aims to reduce the average waiting time for requests by adjusting deployments in real time, considering both resource availability and device mobility. The study effectively models the heterogeneous cloud-edge continuum, accommodating the resource variability inherent in such environments. However, it assumes a linear chain invocation pattern for the MSA application, which limits its adaptability to more complex invocation patterns commonly found in real-world microservice applications.

3.3.3 Summary

Related work demonstrates that heuristic placement algorithms, such as those in [23, 28], often rely on simple criteria like service latency and end-user proximity to derive placement policies for Microservice Architecture (MSA) applications. However, these approaches fail to account for the complex invocation patterns of MSA applications and the underlying heterogeneous computing resources in cloud-edge hybrid environments, leading to potentially suboptimal placement decisions. Optimization-based scheduling methods, like those in [21, 43], are capable of finding optimal microservice placements under well-defined problem schemes. Yet, they require comprehensive and precise modeling of the edge-cloud continuum, and each placement decision involves solving complex optimization problems, resulting in high computational overhead [46]. Consequently, most optimization-based approaches are evaluated only in simulation environments and lack real-world system results.

Alternatively, reinforcement learning has been successfully applied to MSA application placement in several studies [34, 38]. However, its application in the cloud-edge continuum is limited; only a few works [45] have modeled the heterogeneous computing resources in such environments, and these studies do not fully capture the intricate invocation patterns, such as Directed Acyclic Graphs (DAGs), inherent in MSA applications.

Another gap identified in existing research is that most studies focus solely on the initial scheduling of MSA applications. In the dynamic cloud-edge continuum, static

Table 3.1: Comparative Analysis of Related Works

Work	Rescheduling Compliant	Real-World System	Cloud-Edge Continuum Compliant	Network Model	Placement Objectives	Placement Algorithm	Workload	MSA Invocation Pattern
[47]		✓		ND	Cost, Latency	RL	Microservice	Chain
[9]				Connectivity	Cost	RL	Monolithic	DAG
[38]		✓		Distance	RU, Bandwidth	RL	Microservice	DG
[37]				ND	Latency	RL	Monolithic	ND
[7]		✓	✓	Bandwidth	Cost, RU Bandwidth, Latency	RL	Microserivce	DAG
[26]				Bandwidth	Cost, RU Bandwidth, Latency	Meta-Heuristic	Microserivce	DAG
[35]			✓	Connectivity	Cost, Bandwidth	Heuristic	Microservice	DAG
[25]				Bandwidth, Delay	Cost, Latency	Meta-Heuristic	Monolithic	DAG
[40]				Distance	Bandwidth, Latency	Meta-Heuristic	Microserivce	DAG
[27]				Distance	RU, Latency	Meta-Heuristic	Microserivce	DAG
[34]				ND	Deployment Overhead	RL	Microserivce	DAG
[32]				Connectivity	Cost, Latency	Meta-Heuristic	Monolithic	ND
[28]	✓			Delay	Latency	Heuristic	Microservice	DAG
[36]				Bandwidth	Cost, Latency	Mathematical Programming	Microservice	Chain
[24]				Bandwidth	Latency	Mathematical Programming	Microservice	Chain
[39]			✓	Bandwidth, Latency	Cost, Latency	Heuristic	Microservice	DAG
[21]			✓	Latency	Cost, Latency	Meta-Heuristic	Microservice	DAG
[20]	✓	✓		Latency	Latency	Heuristic	Microservice	ND
[45]			✓	Latency	Latency	RL	Microservice	Chain
This work	✓	✓	✓	Bandwidth, Delay	RU, Latency	RL	Microservice	DAG

service placements can become suboptimal as resource availability changes due to factors like computing node failures. Only a few heuristic algorithms [20, 28] are capable of rescheduling microservices to dynamically optimize microservice placement. However, these heuristic methods do not account for the intricate dependencies among microservices or the heterogeneous computing resources characteristic of the cloud-edge continuum.

In this work, we propose a rescheduling algorithm that employs reinforcement learning to derive a placement policy. Compared to existing learning-based approaches, our work models both the MSA application invocation patterns as DAGs and the heterogeneous cloud-edge continuum computing resources. Moreover, the proposed rescheduling algorithm can actively reschedule microservices as resource availability changes, continuously maintaining the optimal placement of MSA applications. Finally, our approach is evaluated in a real-world Kubernetes cloud-edge continuum environment, demonstrating superior performance compared to baseline methods.

Chapter 4

Problem Formulation and RL-based Rescheduling Algorithm Design

To address the challenges in MSA application placement in hybrid cloud/edge environments identified in related work, we present a progressive rescheduling placement strategy. This chapter begins with a system model for MSA applications and cloud-edge resources in Section 4.1. Building on this model, we define the rescheduling problem in Section 4.2. Section 4.3 introduces the reinforcement learning models, followed by a brief overview of relevant reinforcement learning algorithms in Section 4.4.

4.1 System Model

Cloud-edge continuum is a hybrid distributed system environment that integrates computing resources from both the cloud and edge layers. In this architecture, computing nodes in the cloud layer are typically deployed in centralized data centers, providing high resource availability and significant computing power. These characteristics make the cloud layer ideal for applications that require substantial computational resources, enabling efficient execution. However, a key challenge arises because cloud computing nodes are often located far from end users, resulting in higher latency. In contrast, computing nodes in the edge layer are deployed closer to end users, which reduces latency but typically has more limited resources. By combining these two layers, the cloud-edge continuum leverages the strengths of both. On the other hand, the heterogeneity characteristics of computing resources in the cloud-edge continuum make optimizing the end-to-end latency of the MSA application challenging.

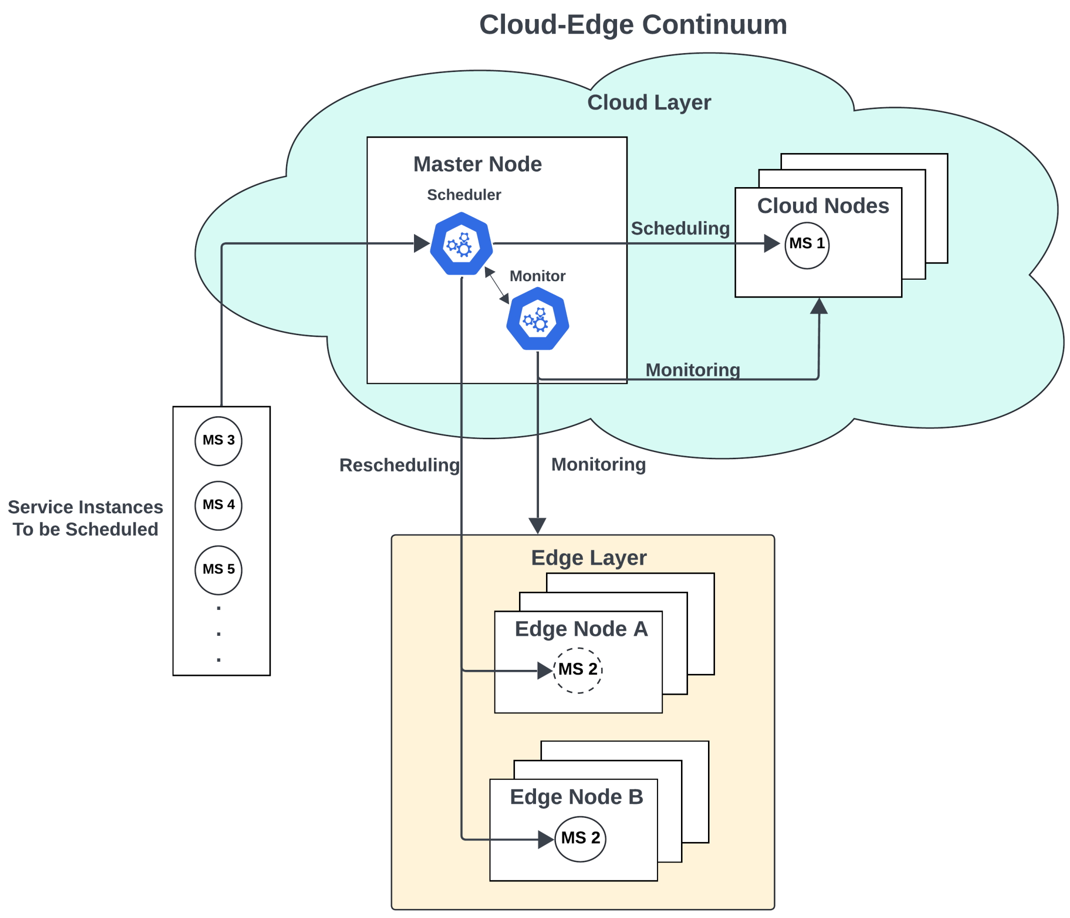


FIGURE 4.1: System Architecture

The placement decision of each service in a microservice architecture (MSA) application within a cloud-edge continuum is crucial. Each microservice may have different characteristics, making it more suitable for deployment in either the cloud or edge layer to achieve optimal performance. Furthermore, microservices within an MSA often have complex internal dependencies, meaning that individual components must be considered together to optimize end-to-end latency.

Figure 4.1 presents the architecture of the proposed system model, where cloud and edge resources are managed collectively by a container orchestrator, forming a unified pool of computing resources. At the core of this architecture lies the scheduler component, which manages the placement of microservices. Initially, the scheduler deploys each service within the MSA application according to a predefined scheduling policy. Once the scheduling is complete, the scheduler continuously monitors both the state of the Cloud-Edge Continuum and the operational status of the MSA applications. Using this real-time information, the scheduler progressively reschedules microservices, aiming to minimize end-to-end latency and enhance overall performance of the MSA applications.

Most existing research [9, 23, 44] focus on the placement policies for scheduling and autoscaling processes. However, relying solely on the combination of Scheduler and Autoscaler does not address all placement challenges in dynamic computing environments. Specifically, the placement decisions made by the Scheduler and Autoscaler may become suboptimal when resource availability in the cloud-edge continuum changes. In such scenarios, the service placement can be improved by rescheduling to different nodes to maintain optimal performance, a task that cannot be achieved by the Scheduler and Autoscaler alone.

To address this issue, we propose a novel approach that formulates the problem as a rescheduling problem. We introduce a Rescheduling Controller that extends the Scheduler's functionality. This controller continuously monitors changes within the cloud-edge continuum and optimizes service placement by progressively rescheduling microservices. This ensures that the system can adapt effectively to dynamic conditions, maintaining optimal placement as resource availability fluctuates.

4.1.1 Application Model

A Microservices Architecture (MSA) application consists of multiple microservices that collaboratively process user requests and handle various tasks. To accurately represent real-world deployments of MSA applications within the cloud-edge continuum, we model each service as potentially having multiple instances to facilitate autoscaling and ensure fault tolerance, as detailed in Section 2.2.1. Each replicated instance of a service is referred to as a **Service Instance,** and a collection of these instances constitutes a **Service**. Every instance within a service i runs the same program and requires identical amounts of CPU and memory resources from the host nodes, denoted as r^i_{cpu} and r^i_{mem}, respectively. The maximum number of replicated instances allowed for a given service i is represented by S^i_{max}, while the current number of running instances within a service i is denoted as S^i_{cur}.

Regarding the load balancing between different services instances, our primary focus is on the placement problem; therefore, we assume that load balancing is performed using a round-robin strategy, which is a widely adopted load balancing strategy that is used by container orchestration platforms like Kubernetes [1] and Marathon [2]. This strategy evenly distributes traffic across all available instances of a service.

The invocation pattern of the modeled application is represented as a Directed Acyclic Graph (DAG), where services may invoke multiple external services. Specifically, each

[1]https://kubernetes.io/
[2]https://mesosphere.github.io/marathon/

service i can invoke a set of external services, and any subset of services that are invoked sequentially is defined as a **Service Group**. Within a service group, services are interdependent and must be invoked in a strict sequence, whereas services belonging to different service groups are independent and can be invoked in parallel. As elaborated in Section 2.2.2 , distinguishing between parallel and sequential invocations is essential, as this differentiation significantly impacts the formulation of the application's end-to-end latency optimization problem.

Symbol	**Description**
S	Number of Services in the Application
N	Number of Nodes in cloud-edge continuum
β	The index set of all services
τ	The index set of all computing nodes
ε	The index set of all running service instances
ψ	The index set of service instances in MSA application
ω	The index set of service groups, $i \in \beta$
E_i	Set of all external services called by a service, $i \in \beta$
S^i_{max}	Maximum number of running instances in service $i \in \beta$
S^i_{cur}	Current number of running instances in service $i \in \beta$
S_{max}	Maximum number of running instances in MSA application.
S_{cur}	Set of all service instances in the running states
L	Set of all node layers, $L = \{cloud, edge\}$
C^i_{cpu}	CPU capacity of a node $i \in \tau$
C^i_{mem}	Memory capacity of a node $i \in \tau$
A^i_{cpu}	CPU availability of a node $i \in \tau$
A^i_{mem}	Memory availability of a node $i \in \tau$
R^i_{cpu}	Total Requested CPU of a node $i \in \tau$
R^i_{mem}	Total Requested Memory of a node i $i \in \tau$
$D_{i,j}$	Latency between two nodes $i \in \tau$, $j \in \tau$
D^i_{user}	Latency between end user and a nodes $i \in \tau$
D_{msa}	End-to-end MSA application latency
D^{ij}_S	Average latency between services i and j. $i \in \beta$, $j \in \beta$
t_i	Internal execution time of running service instance $i \in \psi$
E^i_j	Internal execution time of service instance i running in node j. $i \in \psi$, $j \in \tau$
d_{ij}	Latency between two service instances $i \in \psi$
n_i	Deployed node for a service instance $i \in \psi$
r^i_{cpu}	Requested cpu resource of service instance $i \in \psi$
r^i_{mem}	Requested memory resource of service instance $i \in \psi$
T^i_E	Average internal execution time of service i,
T^i_G	Total Service Invocation Time of Service Group $i \in \omega$
T^i_S	Average service processing time for service $i \in \beta$

Table 4.1: Symbol Table for System Modeling

4.1.2 Cloud-Edge Continuum Model

The Cloud-Edge continuum comprises a set of heterogeneous computing nodes. The CPU and memory capacities of each node i are represented as C^i_{cpu} and C^i_{mem}, respectively, while the available resources at any given time are denoted as A^i_{cpu} and A^i_{mem}. In this dynamic environment, resource availability fluctuates over time, constrained by each node's maximum capacity.

In modern MSA application orchestration platforms like Kubernetes or Mesos, users can specify both the resource requests and resource limits in the service instance specifications.he resource request indicates the minimum amount of CPU and memory a service instance require Ts to be scheduled, while the resource limit defines the maximum amount it can utilize. In practical deployments, this approach can result in scenarios where more service instances are scheduled to one node by satisfying only the minimum resource requests, potentially leading to resource contention and the termination of service instances when resource demands exceed the available supply.

To mitigate these issues, we adopt a more conservative approach by assuming that the resource requests and limits of each service instance are set to the same value. This guarantees that each service instance is allocated a fixed amount of computing resources, effectively preventing resource contention in the computing nodes and minimizing the risk of service instance eviction due to over-commitment. Reflecting this approach in our application model, it implies that for any node in the Cloud-Edge continuum, the total CPU or memory resources requested by all service instances running on that node will not exceed the node's capacity. Specifically, for any node i, the total requested CPU and memory resources, denoted as R^i_{cpu} and R^i_{mem}, must satisfy the following constraints:

$$\sum_{i=1}^{n} R^i_{cpu} \leq C^i_{cpu} \quad and \quad \sum_{i=1}^{n} R^i_{mem} \leq C^i_{mem} \tag{4.1}$$

Here, C^i_{cpu} and C^i_{mem} represent the total available CPU and memory capacity of node i, while n is the number of service instances running on that node.

Another important characteristic of both edge and cloud nodes in the Cloud-Edge continuum is their latency to end users. In this work, we classify each node based on whether it is deployed in the "edge layer", which indicates proximity to the end user, or the "cloud layer", which signifies hosting in remote data centers. In this study, we assume that user devices belong to the edge layer because the edge node is deployed close to the end users. Therefore the latency between end users to any nodes in cloud-edge continuum can be expressed as:

$$D^i_{user} = \begin{cases} D_{e2c} & \text{if node } i \text{ is in cloud layer} \\ \text{or} & \\ D_{e2e} & \text{if node } i \text{ is in edge layer} \end{cases} \tag{4.2}$$

Here, D_{e2c}, D_{e2e} represent latency between edge and cloud layers and latency between edge nodes, respectively. Furthermore, the network latency $D_{i,j}$ between nodes i and j in the cloud-edge continuum also depends on their respective layers, and can be expressed as:

$$D_{i,j} = \begin{cases} D_{e2c} & \text{if nodes } i \text{ and } j \text{ are in different layers (edge and cloud)} \\ \text{or} & \\ D_{e2e} & \text{if nodes } i \text{ and } j \text{ are both in the edge layer} \\ \text{or} & \\ D_{c2c} & \text{if nodes } i \text{ and } j \text{ are both in the cloud layer} \end{cases} \tag{4.3}$$

Here, D_{e2c}, D_{e2e}, and D_{c2c} represent latencies between edge-cloud, edge-edge, and cloud-cloud nodes, respectively.

In this study, we assume that nodes within the same layer exhibit uniform latency characteristics when communicating with nodes from other layers or end users. Although in real-world scenarios, computing nodes within the same data center or geographic region may experience different latencies due to variations in network infrastructure, routing paths, or other factors [4], it is reasonable to assume that computing nodes located in close proximity or within the same region will have similar latency to the end users. This simplification enables us to abstract away some of the inherent complexity in real-world networking while still capturing the essential characteristics of latency in the Cloud-Edge continuum. By modeling latency in this manner, we significantly reduce the problem's complexity, making it more tractable to solve while maintaining a realistic representation of network behavior in such environments.

When calculating end-to-end latency, it is crucial to also consider the internal execution time of a service instance required to process a single request. In the Cloud-Edge Continuum, the average request execution time of a service instance is primarily determined by the computing node on which the service instance is deployed. Variations in factors such as CPU speed, disk I/O capabilities, and memory performance across heterogeneous nodes can lead to differences in execution times for the service's computational tasks [5]. To capture this heterogeneity, we model the different internal execution time of service instances that are running on different nodes.

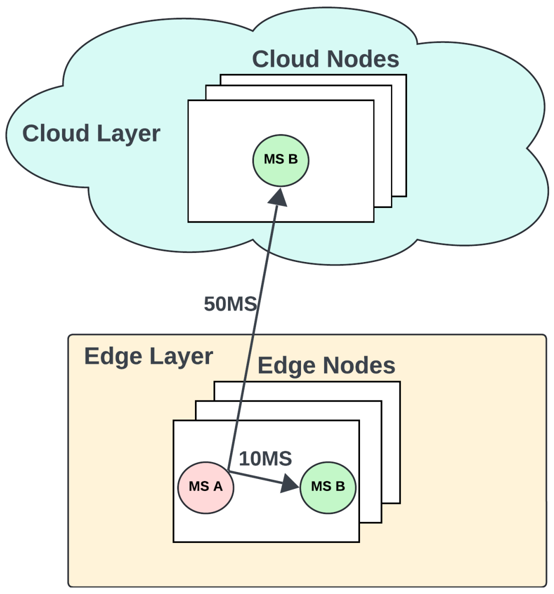


FIGURE 4.2: Latency Difference of Same Service

For any running service instance i, its deployed node is denoted as n_i, therefore we define its service instance execution as $E^i_{n_i}$, which represents the internal execution time of service instance i in its deployed node n_i

4.2 Problem Formulation

We formulate our problem as rescheduling service instances to minimize the end-to-end user request latency. The end-to-end latency comprises the accumulated execution time of services invoked in processing a single user request and the network latency incurred when the request is transmitted between services. Because we modeled the MSA application as a service with multiple replica service instances, the traffic that is sent from one service to another encompasses all network traffic between service instances within those services. This means that the latency between services can vary dramatically depending on the placement of service instances within the same service.

Figure 4.2 illustrates an example scenario where service A communicates with service B, with the service instances of service B deployed across different nodes. In this example, the network latency between nodes in the edge layer is assumed to be 10ms, while the latency between a cloud and edge layer is 50ms. As a result, the overall latency between service A and service B fluctuates depending on the specific placement of service instances, ranging from 10ms to 50ms. Additionally, the execution times of user requests processed by two service instances within service B may vary due to deployment on heterogeneous nodes with differing computational capacities. This heterogeneity introduces variations in execution times for each instance, as nodes with different computing power process requests at different speeds. Consequently, to accurately evaluate the end-to-end latency of the MSA application, it becomes crucial to consider both the network latency between instances of different services and the execution time of service instances within each service.

The internal execution time of a service i on its deployed node n_i is denoted by $E^i_{n_i}$, as specified in Section 4.1.1. For simplicity, we define $t_i = E^i_{n_i}$. Then the average internal execution time of service i, represented as T^i_{exec} is formulated as:

$$T^i_{exec} = \frac{1}{S^i_{cur}} \sum_{j \in \psi_i} t_j \tag{4.4}$$

where ψ_i denote the sets of indexes of service instances in services i . S^i_{cur} denotes the total number of current running service instances in service i.

To model the network latency between services, we start by examining the latency between service instances located on different nodes within the cloud-edge continuum. The latency between any two service instances is determined by the network latency between the nodes hosting them. For any two service instances i and j with their deployed nodes n_i and n_j, we define their latency as $d_{ij} = D_{n_i n_j}$, where $D_{n_i n_j}$ is given in Equation 4.3 and represents the node-to-node latency between n_i and n_j.

When two services that communicate over the network each include multiple service instances, latency fluctuations can occur from request to request due to varying instance pairs being involved, as discussed in Section 4.1.1. To evaluate the expected latency between two services, we derive the average latency across all service instance pairs between the two services. For any services i and j, the sum of all latencies between each pair of service instances is given by:

$$D^{ij}_{sum} = \sum_{k \in \psi_i} \sum_{l \in \psi_j} d_{kl} \tag{4.5}$$

Here, ψ_i and ψ_j denote the set of indexes of service instances in services i and j, respectively. Therefore, $k \in \psi_i$ represents all service instances belonging to service i, and $l \in \psi_j$ represents all service instances in service j.

By dividing the total latency sum by the total number of service instance pairs—which is the product of the number of service instances in each service—we obtain the average latency between services i and j:

$$D^{ij}_S = \frac{1}{S^i_{cur} \cdot S^j_{cur}} D^{ij}_{sum} \tag{4.6}$$

Here, S^i_{cur} and S^j_{cur} represent the numbers of currently running service instances in services i and j, respectively.

Having modeled the average execution time and network latency between services, we now focus on deriving the end-to-end latency for processing a user request in the microservices architecture (MSA) application. We define the average time for a service i to process a user request, starting from the moment the service receives the request, as the average service processing time T^i_S. This processing time consists of two main components: the average internal execution time T^i_E of service i, as given in Equation 4.4, and the average total external service invocation time, which will be modeled in the following sections.

As discussed in Section 2.2.3, when a service instance in a microservices architecture (MSA) invokes external services, the invocations within a service group occur sequentially, whereas invocations to different service groups occur in parallel. For any service group i, the total service invocation time in the service group is denoted as T^i_G , which is calculated by summing the average latency D^{ij}_S between service i and each service j in the group, along with the processing time T^j_S of the invoked external service j:

$$T^i_G = \sum_{j \in g_i} \left(D^{ij}_S + T^j_S \right) \tag{4.7}$$

Here, g_i denotes the set of all service indexes within service group i, so this equation computes the total invocation time for all services j in the group.

The average service processing time T^i_S for service i is recursively defined as the sum of its own internal execution time T^i_E and the maximum total invocation time among all service groups j that are invoked by service i:

$$T^i_S = T^i_E + \max_{j \in \omega_i} T^j_G \tag{4.8}$$

In this expression, ω_i refers to the set of indexes of all service groups invoked by service i.

We define the first service that the end user interacts with as the gateway service g of the MSA application. The average latency between the end users and the gateway service, denoted as $D_{gateway}$, is given by:

$$D_{gateway} = \frac{1}{S^g_{cur}} \sum_{i \in \psi_g} D^i_{user} \tag{4.9}$$

Here, ψ_g represents the set of indexes of service instances in the gateway service g, D^i_{user} is the latency between the end user and service instance i in service g (as formulated in Equation 4.2), and S^g_{cur} is the number of replica service instances of the gateway service g.

The end-to-end latency D_{msa} of the MSA application can then be calculated by adding the average latency between the end user $D_{gateway}$ and the average service processing time of the gateway service T^g_S, which recursively includes the processing times of all subsequent services:

$$D_{msa} = D_{gateway} + T^g_S \tag{4.10}$$

In this work, we design a rescheduling algorithm that dynamically reschedule service instances to adjust their placement within MSA applications in real time. Although our rescheduling process is structured to minimize overhead, as will be discussed in detail in Section 5.4.4, a small overhead remains due to the container orchestrator's need to reroute user requests to newly positioned service instances, resulting in a slight increase in the end-to-end latency of the MSA applications. This rescheduling overhead is modeled as an added latency D_R, contributing to the overall latency. Thus, the final end-to-end latency of the MSA application is represented as:

$$D_{msa} = \begin{cases} D_{gateway} + T^g_S & \text{if no rescheduling process is ongoing} \\ D_{gateway} + T^g_S + D_R & \text{if rescheduling process is ongoing} \end{cases} \tag{4.11}$$

The primary objective of this study is to optimize the end-to-end latency, D_{msa}, for MSA applications by improving service instance placement within the cloud-edge continuum. This challenge, often referred to as the service placement problem, entails finding the optimal placement of services to meet specific performance goals. However, the problem is NP-hard [8, 48], making exact solutions computationally impractical. Consequently, we employ Reinforcement Learning to derive approximate placement solutions aimed at minimizing D_{msa}.

4.3 Reinforcement Learning (RL) Model

Reinforcement Learning is an adaptive approach that optimizes decision-making by enabling the agent to learn through interactions with its environment, receiving feedback in the form of rewards. The primary objective of the RL agent is to maximize cumulative rewards over time, continually refining its decision-making process through trial and error. This adaptive learning process can be formally represented using a Markov Decision Process (MDP) [49], which provides a structured framework for decision-making. An MDP is characterized by the following components:

State space (S): The set of all possible states that are observed by the RL agent.

Action space (A): The set of actions available to the RL agent. In our research, the action is designed to reschedule service instances between computing nodes..

Reward function (R): The immediate feedback the agent receives after taking an action in a specific state. Our approach is primarily optimized for latency, and thus the reward function reflects the change in latency before and after the agent executes a rescheduling decision.

Discount factor (γ): A value between 0 and 1 that determines how much future rewards are valued compared to immediate ones. A lower discount factor makes the agent prioritize short-term rewards, while a higher discount factor encourages the agent to plan for long-term gains.

The objective of the RL agent is to learn a policy $\pi(a|s)$, which defines the probability of taking an action a given a state s, in a way that maximizes the expected cumulative reward:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{N} \gamma^t R(s_t, a_t)\right] \tag{4.12}$$

where $V^{\pi}(s)$ is the value of state s under policy π, and $R(s_t, a_t)$ is the reward obtained at time step t. The goal is to discover the optimal policy π that maximizes this cumulative reward across all states, enabling the RL agent to effectively generate rescheduling action for improved latency and resource optimization.

RL agent learns its policy through interactions with the environment and by receiving rewards. In this work, the RL agent is trained in a simulation environment before being deployed to the real-world testbed. The interaction between the RL agent and simulation environment is shown in Figure 4.3. At the start of each decision-making process, the RL agent is provided with a valid state S_t, representing the current cloud-edge continuum environment and the status of the running service instances of MSA application. The agent then generates a rescheduling action A_t, based on its learned policy $\pi(A_t|S_t)$. The rescheduling action involves selecting a service instance from all the running instances of the MSA application and assigning a target node for rescheduling. Once the action is executed, the application's end-to-end latency may change due to the updated service instance placement. A reward is then given, reflecting the latency change and defined

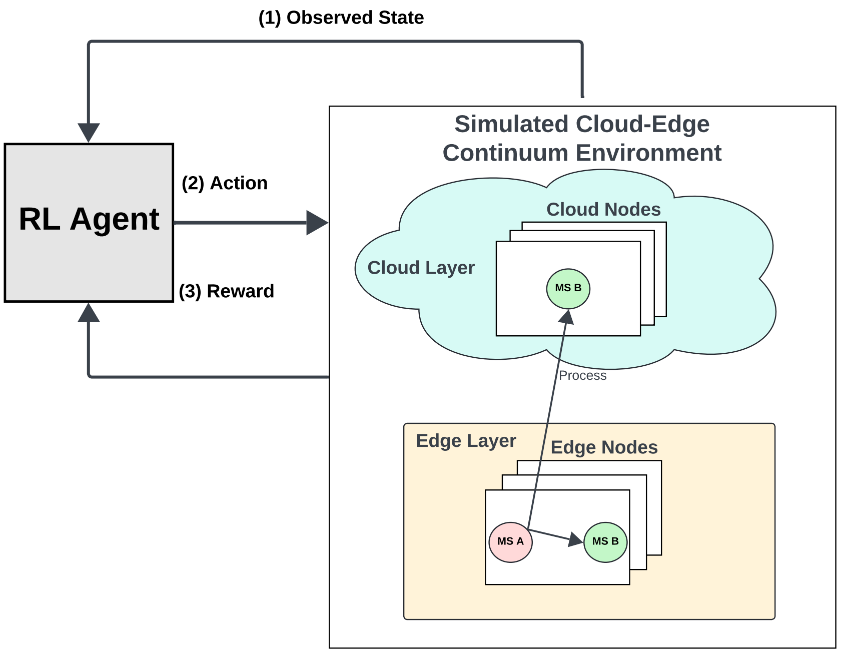


FIGURE 4.3: Interaction Between RL Agent And Environment

by the reward function. Afterward, the agent receives a new observation state S_{t+1}, representing the updated environment.

The following are the key components of the proposed reinforcement learning model:

Episode: At the beginning of each training sequence, the simulation environment resets the state of the cloud-edge continuum. From this reset state, the RL agent interacts with the environment continuously until it selects the "idle" action, marking the end of the sequence.

State: To provide informative states is crucial for enabling our RL agent to learn intricate rescheduling policies. These policies need to be based on the cloud-edge continuum's status and the current state of the running MSA application to execute effective rescheduling actions. For the state of computing resources, the current availability of CPU and memory is represented as vectors $[A^1_{cpu}, A^2_{cpu}, ..., A^n_{cpu}]$ and $[A^1_{mem}, A^2_{mem}, ..., A^n_{mem}]$. Each available resource in computing nodes is constrained within the limits of its respective capacity, denoted as $[C^1_{cpu}, C^2_{cpu}, ..., C^n_{cpu}]$ and $[C^1_{mem}, C^2_{mem}, ..., C^n_{mem}]$. For each pod in the MSA application, its CPU and memory requests are represented in vectors $[r^1_{cpu}, r^2_{cpu}, ..., r^n_{cpu}]$ and $[r^1_{mem}, r^2_{mem} ..., r^n_{mem}]$. Another essential piece of information is which node the pod is deployed on, represented as $[n_1, n_2, ..., n_n]$.

Action Space: The action taken by the RL agent involves selecting a service instance i from the current MSA application and rescheduling it to another node j, represented as an action pair $A(i, j)$.

In addition to the regular rescheduling action, there is a special **Idle** action. This action indicates that, instead of performing any rescheduling, the RL agent chooses not to act. In real-world testbeds, each rescheduling operation incurs an overhead due to the creation and termination of service instances. Therefore, the RL agent must have the ability to halt rescheduling if the current placement is already optimal. To address this, we designed our RL agent to make rescheduling decisions based on the current state of the system.

In the simulation environment, when the idle action is selected, the current training episode ends, and the environment resets for the next episode. In a real-world cloud-edge continuum, selecting the idle action causes the RL agent to pause for a period before starting the next monitoring iteration. The actions taken by the RL agent can be formally expressed as:

$$A = \begin{cases} A(i,j)\ i \in \varepsilon, j \in \tau & \text{if rescheduling is required} \\ Idle & \text{don't perform any rescheduling action} \end{cases} \tag{4.13}$$

Here, $i \in \varepsilon$ represent the set of all service instance indices, and $j \in \tau$ represent the set of indices for all computing nodes in the cloud-edge continuum. The total number of possible rescheduling actions is calculated as $S_{max} \times N$, where S_{max} denotes the maximum number of service instances in MSA applications and N represents the total number of nodes in the cloud-edge continuum. Including the idle action, the RL agent has $S_{max} \times N + 1$ discrete actions available.

When an action is taken by the RL agent, the action should meet certain constraints to be considered as valid action. An action $A(i,j)$ is deemed valid only if the following constraints are satisfied:

$$r_i = true \tag{4.14}$$

$$A^i_{mem} > r^i_{mem} \tag{4.15}$$

$$A^i_{cpu} > r^i_{cpu} \tag{4.16}$$

Here, r_i indicates the whether the current service instance i is in the running state. Therefore, Equation 4.14 ensures that only service instances in the *running* state can be rescheduled. Equations 4.15 and 4.16 specify that the target node must have sufficient available CPU and memory resources to accommodate the requested resources of the service instance.

Reward function: The RL agent learns policies by interacting with the environment. To align the learning process with the objective of minimizing end-to-end application latency, we embed this goal into the reward design. The formal definition of end-to-end latency, D_{msa}, is provided in Equation 4.11. After the RL agent performs a rescheduling action, the end-to-end latency may change. We formulate the first part of the reward function as the difference between the latency before and after a rescheduling action:

$$Reward_D = D_{msa}^{before} - D_{msa}^{after} \tag{4.17}$$

where D_{msa}^{before} and D_{msa}^{after} represent the MSA application end-to-end latencies before and after the rescheduling action, respectively. The intuition behind this is straightforward: if the latency improves, a positive reward is given; if it worsens, a negative reward is applied. The size of the reward or penalty is proportional to the change in latency—larger improvements yield higher rewards, while significant increases lead to heavier penalties. The difference between the latency before and after the rescheduling action captures this heuristic, rewarding the agent based on latency improvement or penalizing it for degradation.

This reward is also well-suited for scenarios in which the agent must first reschedule service instances in a way that temporarily worsens latency, but later performs additional actions to achieve a globally minimized latency. In such cases, even though intermediate steps are penalized, the accumulated reward from the entire sequence of actions reflects the overall latency improvement. Thus, the reward system incentivize the agent to take steps that may involve short-term trade-offs but ultimately lead to better long-term performance.

Another key heuristic in the reward function relates to the number of rescheduling actions required to achieve the lowest latency. As mentioned in the action space design, we designed an Idle action to ensure that the RL agent can stop when there is no further potential for reducing application latency. Therefore, an RL agent that takes fewer rescheduling steps and stops when appropriate should receive a higher reward, promoting the learning of a more efficient policy. To encourage this behavior, we introduce a cost penalty for each action, denoted as $Penalty_{cost}$. This penalty reflects the cost incurred by the RL agent for rescheduling a service instance, motivating it to minimize the steps needed to reach an optimal placement. It is crucial to set this penalty carefully; if it is too high, the RL agent may opt for Idle actions too early, limiting exploration and preventing adequate coverage of the state and action space necessary for convergence.

As discussed in the action space design, the RL agent can produce invalid actions during training. In our problem, valid rescheduling actions are sparsely distributed across the action space. Depending on the current state, the target node in the action may have a high probability of insufficient resources, or the service instance might not be running because there are limited replicated instances. To train the RL agent effectively, ensuring it learns to make valid actions, we impose a significant penalty $Penalty_{invalid}$ for each invalid action.

As elaborated further in section 4.4, we train two different RL agents: Deep Q-Learning (DQN) and Proximal Policy Optimization (PPO). For the PPO agent, we apply an invalid action mask [50] to its policy network, effectively masking out invalid actions during both the training and inference processes. This ensures that the PPO agent is trained exclusively with valid actions. Consequently, no penalty for invalid actions is required for PPO. Therefore, $Penalty_{invalid}$ is applied only to the DQN RL agent. The final reward can be formulated as:

$$Reward\_dqn = \begin{cases} Reward_D + Penalty_{cost} & \text{if action is valid} \\ Penalty_{invalid} & \text{if action is invalid} \end{cases} \tag{4.18}$$

$$Reward\_ppo = Reward_D + Penalty_{cost} \tag{4.19}$$

Example of Episodes of the Proposed RL Model: Figure 4.4 illustrates two episode workouts of the proposed RL agent interacting with the environment and the associated state changes. In the first episode, the agent terminates due to performing an invalid action, while in the second episode, it concludes with the agent generating an ”Idle” action. For simplicity, the figure displays only the states of pod placement and the CPU availability of nodes, along with their changes.

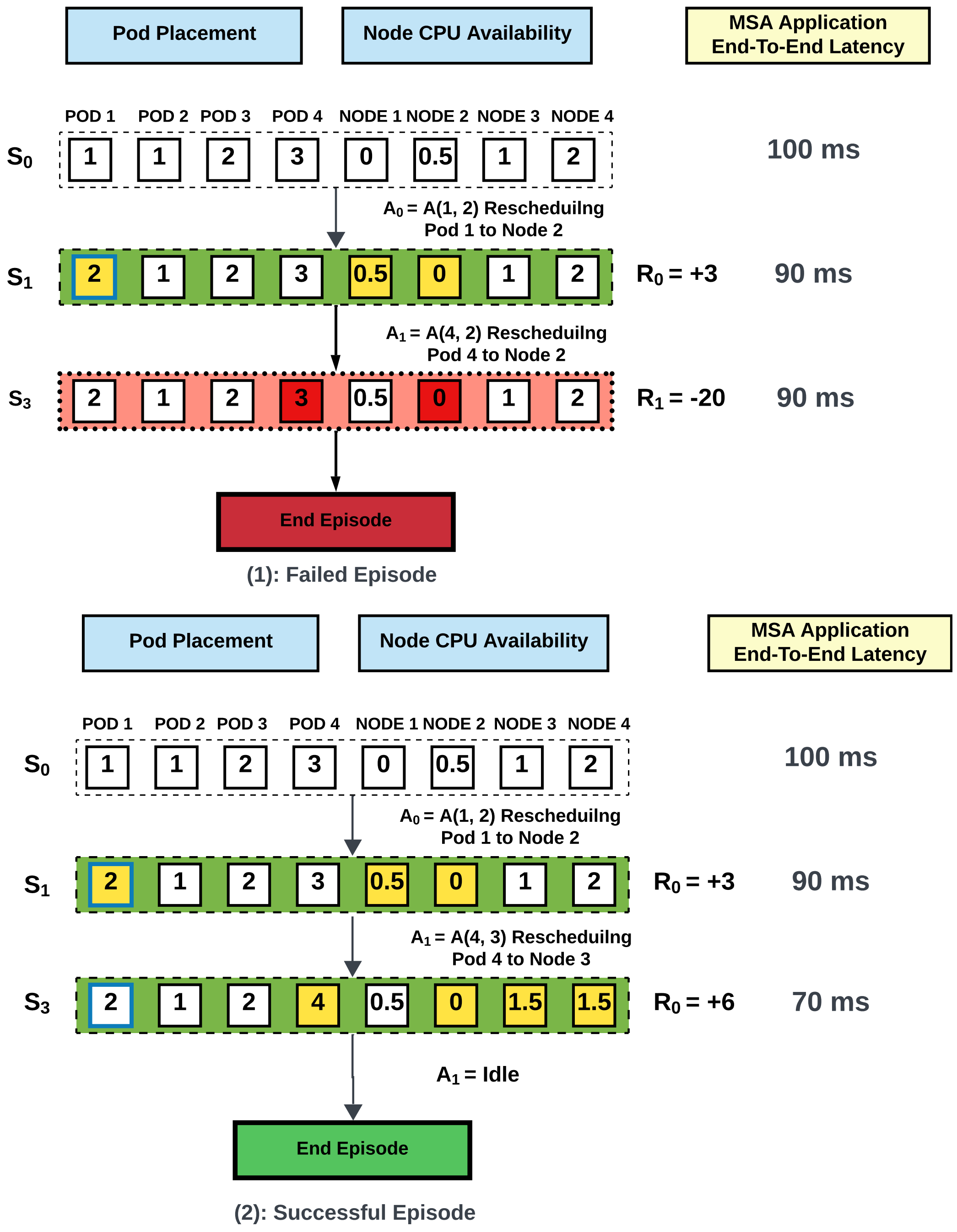

Pod Placement
Node CPU Availability
MSA Application End-To-End Latency
POD 1 POD 2 POD 3 POD 4 NODE 1 NODE 2 NODE 3 NODE 4
S0
1 1 2 3 0 0.5 1 2
100 ms
A0 = A(1, 2) Rescheduilng Pod 1 to Node 2
S1
2 1 2 3 0.5 0 1 2
R0 = +3
90 ms
A1 = A(4, 2) Rescheduilng Pod 4 to Node 2
S3
2 1 2 3 0.5 0 1 2
R1 = -20
90 ms
End Episode
(1): Failed Episode
Pod Placement
Node CPU Availability
MSA Application End-To-End Latency
POD 1 POD 2 POD 3 POD 4 NODE 1 NODE 2 NODE 3 NODE 4
S0
1 1 2 3 0 0.5 1 2
100 ms
A0 = A(1, 2) Rescheduilng Pod 1 to Node 2
S1
2 1 2 3 0.5 0 1 2
R0 = +3
90 ms
A1 = A(4, 3) Rescheduilng Pod 4 to Node 3
S3
2 1 2 4 0.5 0 1.5 1.5
R0 = +6
70 ms
A1 = Idle
End Episode
(2): Successful Episode

FIGURE 4.4: Example of State Transitions in Proposed RL Model

In both cases, there are four service instances $\{SI_1, SI_2, SI_3, SI_4\}$ deployed across four nodes $\{Node_1, Node_2, Node_3, Node_4\}$. We assume that each service instance requests 0.5 units of CPU resources. Initially, SI_1 and SI_2 are placed on $Node_2$, while SI_3 and SI_4 are placed on $Node_3$ and $Node_4$, respectively. The initial latency of the MSA application is 100 ms.

In the first episode, the RL agent first produces the action $A_0 = A(1, 2)$, which reschedules SI_1 to $Node_2$. This results in a placement state transition for SI_1 from $Node_1$ to

$Node_2$. Consequently, the CPU resource availability of $Node_1$ increases from 0 to 0.5 due to the release of resources by SI_1, while the CPU availability of $Node_2$ decreases from 0.5 to 0. After this rescheduling, the application latency decreases from 100 ms to 90 ms. Based on the reward function defined in Equation 4.18, and setting the rescheduling cost penalty $Penalty_{cost} = 5$, the RL agent receives a reward calculated as: $Reward = 100 - 90 - 5 = 5$. Following this, the RL agent attempts an invalid action $A_1 = A(4, 2)$, aiming to reschedule SI_4 to $Node_2$, which at this point lacks sufficient CPU resources. As a result, the episode ends, and the agent receives an invalid action penalty $Penalty_{invalid}$, set to -100.

In the second episode, the RL agent begins by performing the same initial action as in the first episode, leading to state S_1. It then issues another valid action $A_1 = A(4, 3)$, rescheduling SI_4 to $Node_3$. This results in state changes for SI_4's placement from $Node_4$ to $Node_3$, the CPU availability of $Node_3$ decreasing from 1 to 0.5, and the CPU availability of $Node_4$ increasing from 1.5 to 2.0. This placement further reduces the application latency from 90 ms to 70 ms, yielding a reward of: $Reward = 90-70-5 = 15$ After reaching state S_2, the RL agent produces an "Idle" action, which ends the episode and resets the environment.

4.4 RL-based Rescheduling Algorithms

In this work, we employ two distinct RL agents to address the rescheduling problem. The first agent is DQN (Deep Q-Learning), a value-based RL agent that predicts the potential value based on the current state and the action taken [51]. The second agent is PPO (Proximal Policy Optimization) [6], a policy gradient approach that enables the RL agent to directly learn the policy itself. We employ these two RL agents because our rescheduling problem models the action space as discrete, and these agents are particularly effective for such environments. DQN and PPO also leverage neural networks to approximate value functions and policy functions respectively, enabling them to handle complex and high-dimensional state spaces more efficiently.

4.4.1 Deep Q-Learning

DQN is a value-based method focused on learning the action-value function $Q(s, a)$, which estimates the expected cumulative reward after taking action a in the state s. The agent's policy is indirectly derived by selecting the action that maximizes $Q(s, a)$. Deep Q-Learning (DQN) is an extension of the basic Q-learning algorithm designed to handle large state and action spaces by using a neural network to approximate the action-value function. Instead of storing explicit $Q(s, a)$ values for all state-action pairs, DQN uses a neural network with parameters θ to predict the Q-values for possible actions. The training goal is to minimize the Temporal Difference (TD) error, which represents the difference between the predicted Q-value and the target value. The TD error δ_t is defined as:

$$\delta_t = y_t - Q(s_t, a_t; \theta) \tag{4.20}$$

where y_t is the target value computed as the sum of the immediate reward and the maximum discounted Q-value for the next state. The loss function to minimize the TD error is given by:

$$\mathcal{L}(\theta) = \mathbb{E}\left[(y_t - Q(s_t, a_t; \theta))^2\right] \tag{4.21}$$

The network parameters θ are updated using gradient descent as follows:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta) \tag{4.22}$$

where α is the learning rate, and $\nabla_\theta \mathcal{L}(\theta)$ represents the gradient of the loss function concerning the parameters θ. This update step progressively reduces the loss, bringing the predicted Q-values closer to the target values and thus helping to derive an effective policy. Since DQN learns to estimate the potential rewards of a state-action pair, it can utilize a replay buffer [52] that stores past state-action pairs along with their associated rewards. By randomly sampling from this buffer during training, DQN improves training stability and reduces the risk of overfitting to recent experiences.

4.4.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a robust reinforcement learning (RL) algorithm that builds upon the principles of policy gradient methods. It directly learns a policy π, represented as a neural network that outputs a probability distribution over possible actions. In traditional policy gradient approaches, the policy is updated by maximizing the expected cumulative reward. The basic update rule for the policy network is:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \mathbb{E}\left[\sum_{t=0}^{N} \gamma^t R(s_t, a_t)\right] \tag{4.23}$$

where θ denotes the policy parameters, and $R(s_t, a_t)$ represents the reward obtained from taking action a_t in state s_t. This update process increases the likelihood of actions that yield higher rewards. PPO refines this approach by incorporating the actor-critic framework, which consists of two neural networks: the actor-network and the critic network. The actor-network, expressed as $\pi(a|s;\theta^\pi)$, is the policy network that outputs the action distribution, while the critic network, represented as $V(s;\theta^V)$, estimates the expected reward from a given state, similar to the Q-value function $Q(s,a)$ in DQN agents. Essentially, the actor-critic framework combines the strengths of both policy-gradient and value-based methods.

During training, the critic network $V(s;\theta^V)$ is used to derive an advantage function $A(s_t,a_t)$, which measures how much better an action is compared to the critic's value estimate. The advantage function is defined as:

$$A(s_t,a_t) = R(s_t,a_t) + \gamma V(s_{t+1};\theta^V) - V(s_t;\theta^V) \tag{4.24}$$

Here, $V(s_{t+1};\theta^V)$ is the estimated value of the next state, and $V(s_t;\theta^V)$ is the value of the current state. The advantage function essentially compares the actual outcome (reward and the value of the next state) to the predicted outcome (value of the current state). Using the advantage, the actor policy parameters θ^π are updated as follows:

$$\theta^\pi \leftarrow \theta^\pi + \alpha \nabla_{\theta^\pi} \log \pi(a_t|s_t;\theta^\pi) A(s_t,a_t) \tag{4.25}$$

In this equation, $\nabla_{\theta^\pi} \log \pi(a_t|s_t;\theta^\pi)$ represents the gradient of the log-probability of taking action a_t. Compared to traditional policy gradient approaches, this update method incorporates the advantage function, which helps to emphasize actions that lead to better outcomes than expected, thereby guiding the policy more effectively towards higher reward actions.

Simultaneously, the critic network θ^V is updated to minimize the Temporal Difference (TD) error, a process similar to the Q-value update in DQN:

$$\theta^V \leftarrow \theta^V - \beta \nabla_{\theta^V} \left(R(s_t,a_t) + \gamma V(s_{t+1};\theta^V) - V(s_t;\theta^V) \right)^2 \tag{4.26}$$

The actor-critic architecture employed by PPO improves the sample efficiency of the learning process. By incorporating the advantage function into the policy update, the value function guides the exploration of the RL agent more effectively. Another key feature of PPO is its clipping mechanism, which prevents excessively large updates to the policy network [53]. This feature stabilizes the learning process by ensuring that the updates do not lead to drastic changes in the policy, which could otherwise destabilize the training and negate the progress made by the agent.

4.4.3 Handling Invalid Rescheduling Actions

As discussed in section 4.3, a rescheduling action is considered invalid if the selected service instance is not running, or if the target node lacks sufficient resources to provision the rescheduled service instance. Traditional RL training handles invalid actions by applying a heavy penalty whenever the RL agent produces an invalid action. While this approach can eventually teach the agent to generate valid actions, it often requires much longer training episodes. If valid actions are sparse within the action space, the RL agent may struggle to converge effectively.

Huang et al. [50] proposed an invalid action mask mechanism that can be efficiently applied to policy-gradient RL agents. In the policy-gradient approach, the policy network outputs unnormalized scores (logits) for each action, which are then passed through a softmax function to produce a probability distribution over the actions. To mask out invalid actions, an action mask is first generated as:

$$Mask(i) = \begin{cases} 0 & \text{if action } i \text{ is valid} \\ -\infty & \text{if action } i \text{ is invalid} \end{cases} \tag{4.27}$$

The mask is applied to the logits of the policy network during both the training and inference phases of the RL agent. When applied in the training phase, the mask prevents the RL agent from sampling invalid actions, effectively enabling it to explore only within the valid action space and thus facilitating faster convergence. During the inference phase, the invalid mask can be used to eliminate the possibility of the RL agent generating invalid actions.

In this work, we applied the action mask to the PPO agent, while the DQN agent was trained without a mask, as this mechanism has only been proven effective with policy-gradient agents. Instead, the DQN is trained by applying an invalid action penalty $Penalty_{invalid}$ as defined in Equation 4.18

Chapter 5

System Implementation

In this thesis, we have successfully implemented a comprehensive system that integrates a Reinforcement Learning (RL)-based rescheduling algorithm and evaluates its effectiveness in a real-world Kubernetes testbed. Figure 5.1 outlines the workflow of the proposed system. Initially, we developed three sample Microservice Architecture (MSA) applications with diverse service invocation patterns using μ**Bench** [54], a widely recognized toolkit for building benchmark MSA applications. These benchmark applications were then strategically deployed in a **Kubernetes cloud-edge continuum testbed**, where we gathered application profiling data. This data was subsequently imported into our proposed RL simulation environment, **CEEnv**, allowing for the accurate simulation of end-to-end latency across various service placements within the cloud-edge continuum. Through this simulation environment, we trained our RL agents and reached convergence. Our work extended beyond simulation: we developed a **rescheduling plugin**, which incorporates trained RL agents to enable rescheduling functionality within the real-world Kubernetes testbed. Finally, we redeployed the benchmark applications in the testbed and utilized the RL agent with the rescheduling plugin to assess its performance in reducing the end-to-end latency of MSA applications.

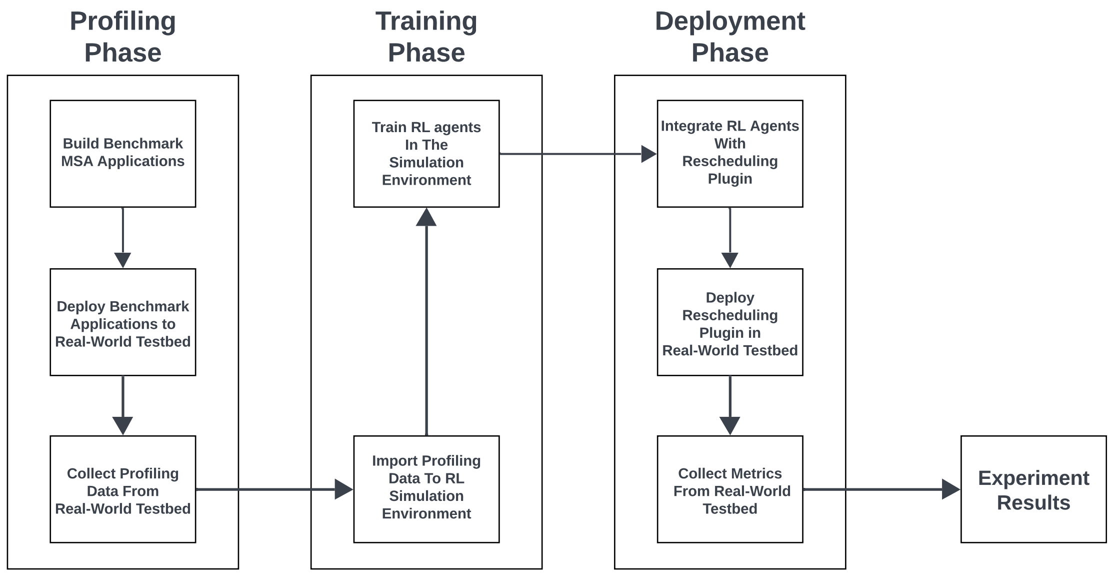

Profiling Phase
Build Benchmark MSA Applications
Deploy Benchmark Applications to Real-World Testbed
Collect Profiling Data From Real-World Testbed
Training Phase
Train RL agents In The Simulation Environment
Import Profiling Data To RL Simulation Environment
Deployment Phase
Integrate RL Agents With Rescheduling Plugin
Deploy Rescheduling Plugin in Real-World Testbed
Collect Metrics From Real-World Testbed
Experiment Results

FIGURE 5.1: Workflow of the implemented system. First, we construct a benchmark MSA application and collect profiling data in a real-world testbed. This data is then used to train an RL agent in the CEEnv simulation environment. Finally, the trained agent is deployed via the rescheduling plugin and evaluated in the testbed.

In this chapter, we begin by introducing the Microservice Architecture (MSA) application we constructed, as detailed in Section 5.1. Following that, we present the implementation of the Reinforcement Learning (RL) simulation environment in Section 5.2, which serves as the foundation for our subsequent research. Leveraging this environment, we discuss the training process of the underlying RL agent in Section 5.3. Subsequently, we detail the implementation of the rescheduling plugin in Section 5.4, highlighting how it integrates with the RL agent to optimize microservice placement. Finally, in Section 5.5, we describe the MSA application profiler used in this work to collect MSA application profiling data and metrics for evaluation.

5.1 Microservice Architecture Application

In this work, we utilize Kubernetes [1] to orchestrate the deployment of microservice architecture (MSA) applications across heterogeneous computing nodes within the cloud-edge continuum. Kubernetes, as the leading container orchestration platform, provides extensive configurability, enabling flexible deployment strategies that are essential for testing our reinforcement learning (RL)-based rescheduling module.

Several existing benchmarks facilitate MSA research in Kubernetes environments. For example, Istio's Bookinfo [2] demo serves as a widely-used MSA application benchmark,

[1]https://kubernetes.io/
[2]https://istio.io/latest/docs/examples/bookinfo/

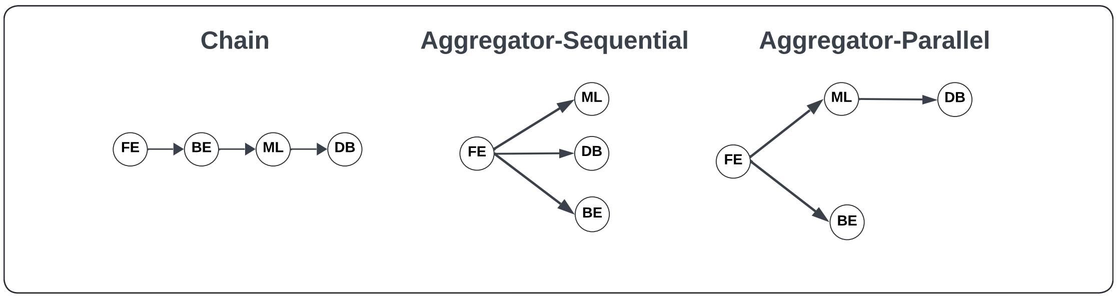


Figure 5.2: Constructed benchmark MSA applications with varying service invocation patterns and orders. In **Chain**, services call each other sequentially. In **Aggregator-Sequential**, the Front-End service calls ML, DB, and Back-End services in sequence, while in **Aggregator-Parallel**, the Front-End calls ML and Back-End services in parallel.

while DeathStarBench [3] , developed for social media microservices, has been extensively adopted in cloud-based MSA studies. However, these benchmarks often target specific workloads, resulting in microservices that have limited configurability, especially concerning invocation patterns, computational complexity, and other vital parameters.

To address this limitation, our work compares and evaluates the performance of our rescheduling policy by deploying MSA applications with diverse invocation patterns. We chose μBench [54] as our primary toolset due to its ability to declaratively construct microservice benchmarks, seamlessly integrate with Kubernetes, and export metrics via Prometheus [4], which is a widely adopted application metrics database. This integration facilitates the collection of detailed profiling data, such as internal execution times and external invocation latencies, which are critical for simulation and analysis.

Using μBench, we designed three MSA applications, each reflecting a different invocation pattern. Figure 5.2 illustrates the constructed MSA applications. In the "chain" application, each microservice calls only one external service, forming a linear invocation pattern. The "aggregator-sequential" application features a front-end service that sequentially invokes multiple other services, while the "aggregator-parallel" application allows the front-end to call two external services concurrently. All applications share the same set of microservices but differ in how they interact, emphasizing different invocation patterns. Each MSA application comprises four services, with each service supporting up to five replicas, resulting in a maximum of 21 pods per application. An additional client pod is deployed on specific client nodes to simulate end-user requests directed to the front-end service. Kubernetes manages load balancing within the cluster, ensuring that requests are evenly distributed across all replicated pods of a service.

[3]https://github.com/delimitrou/DeathStarBench
[4]https://prometheus.io/

TABLE 5.1: Microservice Configuration

	client	front-end	ml	back-end	db
pi calculation complexity	1	200	400	200	100
iteration of pi calculation	1	2	8	4	1
memory writing bytes(KB)	0	1000	2000	1000	100
disk writing	0	1	1	1	100

Microservices within our workload are defined and instantiated using μBench, which allows precise control over the stress on computing resources for each user request. It's important to note that the names assigned to these microservices are used primarily for ease of identification, based on the types of resource stress (CPU, memory, disk) they impose, rather than their specific functionalities. These configurations help mimic the heterogeneous task characteristics typically found in real-world applications.

A microservice configured through μBench can stress CPU resources by calculating digits of π up to a specified complexity, stress memory by loading a specific amount of data, and stress disk by writing a defined number of bytes. Detailed parameters for each microservices resource usage are provided in Table 5.1. In our configuration, the **front-end** microservice is lightweight, primarily handling user requests and routing them to other services. The **back-end** service has a moderate workload, stressing both CPU and memory resources. The **ml** service is computationally intensive, imposing high demands on CPU, memory, and disk. Lastly, the **db** service focuses on disk-intensive operations, akin to typical database workloads.

5.2 RL Environment Design and Implementation

Training Reinforcement Learning (RL) agents require extensive interaction with an environment to receive feedback [55], which is essential for learning effective policies. The accuracy with which the environment simulates real-world conditions directly influences the quality of the learned policies; higher fidelity simulations lead to superior policy performance. Consequently, numerous studies [9, 33, 34] leveraging RL to address placement problems have opted to train their agents directly within real-world testbeds.

However, the training of RL agents often demands a vast number of interactions with the environment [55]. In the context of our research, achieving a robust RL policy may require hundreds of such interactions. Within Kubernetes, each pod scheduling operation can take several seconds, resulting in an exponential increase in training time as the problem scale expands. To mitigate this challenge, many studies [23, 26, 31], employ simulation environments like CloudSim [56], iFogSim [57], or EdgeCloudSim [58] to train RL agents for cloud and edge computing problems. These simulators primarily focus on modeling cloud and edge computing resources but often fall short of accurately simulating the placement behaviors of microservice applications (MSAs) within modern container orchestration platforms such as Kubernetes.

5.2.1 The Proposed Reinforcement Learning Environment

In this study, we introduce **CEEnv**, a reinforcement learning cloud-edge continuum simulation environment designed for training RL agents to address the MSA application placement problem. At a high level, the RL agent is initially trained within CEEnv to learn a policy for rescheduling service instances in the MSA application. Once trained, the agent is deployed in a real-world cloud-edge continuum—specifically, a Kubernetes testbed—to execute actual rescheduling actions. As discussed in Section 4.3, the RL agent learns its policy by interacting with the environment and receiving rewards that evaluate the effectiveness of its actions. In our rescheduling problem, the difference in end-to-end latency of the MSA application before and after a rescheduling action is a crucial element of the reward design. Therefore, the simulation environment must be capable of modeling MSA application end-to-end latency based on the placement of service instances within the cloud-edge continuum.

CEEnv simulates the end-to-end latency of MSA applications under various configurations of cloud-edge continuum resources and different MSA application workloads. This capability allows the RL agent to learn effective rescheduling policies that can be directly applied to a real-world Kubernetes cloud-edge continuum testbed.

Figure 5.3 illustrates the overall architecture of CEEnv. The simulation architecture adopts a hierarchical software design, with each component implemented as a class that encapsulates its respective functionality and data, ensuring modularity and maintainability.

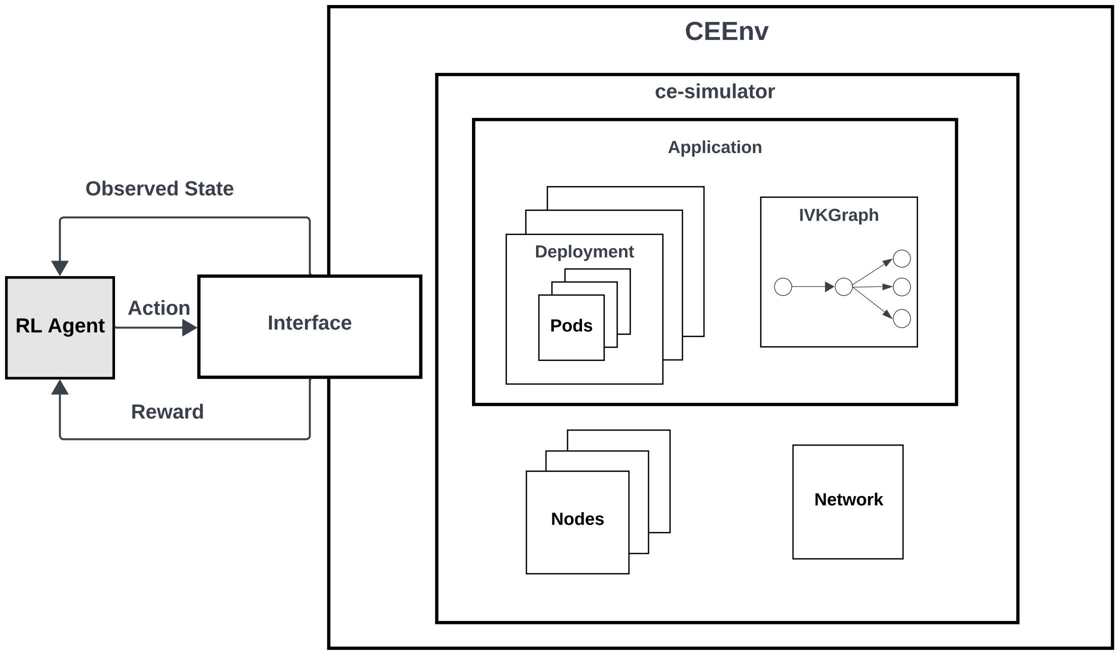


FIGURE 5.3: Architecture of CEEnv. The outer layer provides the interface used by the RL agent framework to train RL agents. CEEnv includes the ce-simulator component, which manages the data structures for the cloud-edge continuum resources and the MSA application, including its invocation pattern.

The outermost layer is the **rl-env** layer, which implements the interface required for the RL agent to interact with CEEnv. Key operations include **reset()**, which resets the state of CEEnv, and **step()**, which applies an action and returns rewards along with the resulting observation state. The interface provided by rl-env is compatible with Gymnasium [5], a widely used RL environment framework, known for its robust support for stable-baselines3 footnotehttps://stable-baselines3.readthedocs.io/en/master/, a popular RL agent library. In this study, stable-baselines3 are utilized to train the RL agents.

The core component of **rl-env** is the **ce-simulator**, which plays a pivotal role in simulating the end-to-end latency of MSA applications. The ce-simulator provides interfaces for managing the placement of MSA applications, including **schedule()**, which schedules service instances of MSA application to the simulated cloud-edge continuum computing resources, and **reschedule(target_node_id, pod_id)**, which reschedules a currently running pod to the specified target node.

To implement the core simulation functionality of **ce-simulator**, several classes have been developed to simulate not only the MSA applications and the invocation patterns between services but also the underlying cloud-edge computing resources. The **ce-simulator** is designed to align with Kubernetes' architecture for deploying MSA

[5]https://gymnasium.farama.org/

applications, enabling seamless definition and deployment of MSA applications across both simulation environments and Kubernetes testbeds. Specifically, Kubernetes organizes MSA applications using a hierarchical structure. At the lowest level, the **Pod** is the smallest computing unit, representing a service instance in our model. A **Deployment** is then used to manage a set of replicated pods, acting as a complete service that balances the load. An MSA application can include multiple Deployment resources. The **ce-simulator** adopts this design, utilizing the following classes to simulate MSA applications:

Pod: Each service instance in the ce-simulator is represented as a **Pod** object. This object records the node where the pod is hosted, as well as the CPU and memory resources it requests from the host node.

Deployment: A Deployment defines a set of replicated pods for a specific service. The **Deployment** object maintains a list of its replicated pods as **Pod** objects, effectively representing the entire set of service instances.

IVKGraph: This object defines the invocation graph of the MSA application as a Directed Acyclic Graph (DAG). Each service invocation is represented as a **Service** object within the graph structure of the IVKGraph. The IVKGraph is essential for calculating the end-to-end latency of the application, which will be elaborated on later.

Application: This class contains metadata about the MSA application, including its name, a list of Deployment objects that represent the microservices of the application, and the IVKGraph, which stores the service invocation patterns.

Along with the classes related to MSA applications, the key classes used to simulate the cloud-edge continuum are composed of the following:

Node: The Node class stores metadata about a computing node within the cloud-edge continuum. It records the availability of CPU and memory resources, the layer (cloud or edge) to which it belongs, and its type. The type of information classifies a collection of computing nodes that share similar computational power. As discussed in Section 4.1.2 We assume that if a pod runs on different nodes of the same type, it will experience similar execution times.

Network: This class encapsulates network latency information within the cloud-edge continuum. As discussed in Section 4.1.2, we model inter-node latency based on whether the nodes are hosted in the cloud or edge layer. Therefore, the Network object stores data on latency between cloud, edge, and user layers, as well as latency between nodes within the same layer.

5.2.1.1 Profiling Data From Real-Word Environment

To capture the characteristics of the cloud-edge continuum, application profiling data must be collected from a real-world testbed and integrated into CEEnv. Specifically, two types of data need to be gathered. The first is the average latency between nodes across the cloud, edge, and user layers. This data is essential for simulating node-to-node latency within the cloud-edge continuum, as well as the latency between nodes and end users. The second type of data is the average execution time for the pods in a service running on specific types of computing nodes. This information will be used to calculate the overall end-to-end latency based on the types of nodes where each pod in the MSA application is deployed.

These two types of profiling data can be efficiently collected using modern monitoring and tracing tools, such as Istio or Jaeger, with minimal effort. The collected data is then fed into the simulation environment, along with the definitions of the MSA application and cloud-edge continuum, through a JSON file.

5.2.1.2 Custom Configurations of CEEnv

CEEnv allows users to custom-define heterogeneous cloud and edge computing nodes as well as MSA applications. In our work, we first set up a Kubernetes cloud-edge continuum testbed and created MSA workloads with different invocation patterns. These configurations were then mirrored in CEEnv to reflect the testbed setup and the MSA application workloads we created.

To map a real-world cloud-edge continuum in CEEnv, users must first identify different types of heterogeneous computing resources in the actual environment. As discussed in Section 4.1.2, nodes in the cloud-edge continuum are grouped into types based on their computing speed, ensuring that the execution time of the same service on nodes of the same type remains similar. For example, in our real-world testbed, we identified three types of computing nodes: **Cloud-A**, **Edge-A**, and **Edge-B**, where their average execution times for the same machine-learning microservice **ml** were 50ms, 69ms, and 90ms, respectively. Based on these performance metrics, we categorized them as different types of nodes.

After defining the node types, specific node specifications are configured, including CPU and memory availability, associated node types, the layers (cloud, edge, or user) in which these nodes are located within the cloud-edge continuum, and the number of nodes with the same specifications.

Alongside the custom definition of cloud/edge computing resources, users can also define the MSA application workload. For each service in the MSA application, users specify essential information such as CPU and memory requirements and the number of replicated service instances. Additionally, as discussed in Section 4.1.1, each service can define a list of Service Groups, where each Service Group contains a sequence of external services to be called sequentially, while services in different Service Groups are called in parallel. This approach allows users to create MSA applications with complex invocation patterns, making CEEnv suitable for simulating real-world MSA applications.

5.2.1.3 Simulating MSA Application End-To-End Latency

The core functionality of CEEnv lies in its ability to simulate the end-to-end latency of MSA applications based on their placement across cloud and edge computing resources. To achieve this, MSA application profiling data and cloud-edge continuum network profiling data are fed into CEEnv for the ce-simulator to evaluate pod-to-pod latency and pod internal executing time. User-provided configurations for the cloud-edge continuum and the MSA application are then used to generate the **IVKGraph**, which is a crucial graph structure that preserves the invocation pattern of the MSA application.

Many existing works [7, 9, 25], model the invocation pattern of MSA applications as a simple Directed Acyclic Graph (DAG), where each service is represented as a vertex, and all outgoing directed edges point to the external services it calls. Although this structure can illustrate the interactions between microservices, it fails to capture the specific order of external service invocations. In CEEnv, the **IVKGraph** is also structured as a DAG; however, it enhances the representation by storing services' calling external services within a list of **Service Groups** as discussed in Section 4.1.1. Each service in a Service Group is invoked sequentially, reflecting a dependency order, while different Service Groups are executed in parallel. This approach allows CEEnv to accurately model complex invocation patterns, making it more effective for simulating real-world MSA application behaviors.

Figure 5.4 provides an example of the **IVKGraph** structure, which models a basic invocation pattern within a Microservice Architecture (MSA) application. In this instance, Microservice *Service A* is organized into two service groups. In the first service group, *Service A* makes sequential calls to *Service B* followed by *Service C*. In the second group, *Service A* calls *Service D*. These two groups operate in parallel, meaning that *Service A* simultaneously invokes both service group 1 and service group 2.

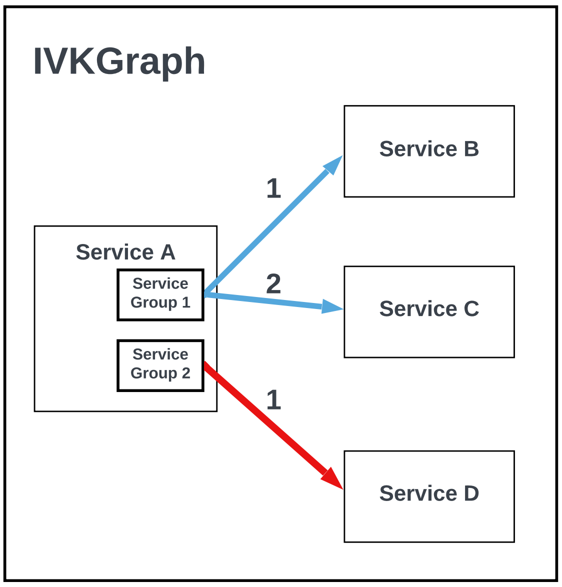


FIGURE 5.4: Structure of **IKVGraph**, which includes 4 services in a graph structure

The construction of the **IVKGraph** ensures that the simulation environment captures both the invocation patterns and the order of service calls, both of which are essential for calculating the end-to-end latency. Once the IVKGraph is generated, CEEnv schedules the MSA application by binding the **Pod** objects within each **Deployment** object to specific **Node** objects. After the scheduling process is completed, the MSA application is considered to be in a running state, and CEEnv proceeds to calculate the end-to-end latency based on the defined service placements.

The calculation of end-to-end latency is a graph traversal problem [59]. The end-to-end latency encompasses all network latencies incurred along the service call graph, as well as the internal execution time required by each pod to process a user request.

The **IVKGraph** stores a **Service** object, which acts as the entry point for the MSA application. The **ce-simulator** begins from this entry Service object and uses a Depth First Search (DFS) algorithm to traverse all Service objects within the IVKGraph.

During the graph traversal, the **ce-simulator** calculates the total service execution time for each service. This total execution time is the sum of the average internal execution time of service replica pods and the average time spent invoking external services. This method ensures that both processing and communication latencies are accurately accounted for in the overall end-to-end latency calculation.

The average internal execution time for a service is calculated based on the average internal execution times of all the pods hosted on different nodes. The MSA application profiling data is used to determine the expected execution time of a pod when it is deployed on a specific type of node. This ensures that the simulation reflects the performance characteristics of different node types within the real-world cloud-edge continuum.

The average external service invocation time for a service i is determined by calculating the maximum invocation time among the services in any of the service groups that service i calls. This is because, according to our definition, services within a Service Group are invoked in parallel.

For each Service Group j, its invocation time is calculated as the sum of the invocation times of all services k within that group, since these services are executed sequentially. The invocation time for service i to call service k is computed as the sum of the average network latency between all pods of service i and service k, plus the average execution time of service k, which is obtained during the DFS traversal process.

The pod-to-pod latency is derived from the network latency information provided by the profiling data, which allows the ce-simulator to simulate network delays between nodes, ensuring a realistic simulation of the end-to-end latency for the MSA application.

By incorporating the invocation patterns and the execution order of services within MSA applications, and by calculating the internal execution times and pod-to-pod latencies using the profiling data, the **ce-simulator** effectively simulates the end-to-end latency of MSA applications. This simulation reflects the performance of real-world MSA deployments within a Kubernetes testbed.

5.3 RL Agent Training

In this work, we employ our custom-developed CEEnv environment to train reinforcement learning (RL) agents for dynamic rescheduling within a cloud-edge continuum. Before initiating the training process, we configure CEEnv to accurately reflect the computing resources available in our Kubernetes testbed. The details of this configuration, including the resource allocation and infrastructure setup, are thoroughly discussed in Section 6.1. To gain insights into the behavior of the microservice architecture (MSA) applications, we conduct a profiling process on three distinct applications: **Chain**, **Aggregator-Parallel**, and **Aggregator-Sequential**. This profiling is achieved through the use of monitoring tools deployed within the Kubernetes cluster, allowing us to gather performance metrics and usage patterns, which will be elaborated in Section 5.5.

For each of these workloads, we train two types of RL agents—Deep Q-Network (DQN) and Proximal Policy Optimization (PPO), which allows us to conduct a comparative analysis of the effectiveness of different RL algorithms in addressing the challenges associated with MSA application rescheduling, providing insights into which strategies offer better performance and adaptability.

CEEnv is adapted to the Gymnasium interface, making it compatible with a wide range of RL agent implementations. We employ the Stable-Baselines3 [6] framework, which supports multiple reinforcement learning algorithms, to train both the DQN and PPO agents. Notably, stable-baselines3 provides a masking mechanism for PPO, as discussed in Section 4.4.3, allowing the agent to explore only valid actions without incurring penalties for invalid ones.

The reinforcement learning (RL) agents adapted to the Stable-Baselines3 framework are trained within CEEnv. In each training episode, the RL agent starts from an initial state of the cloud-edge continuum with a deployed MSA application. It interacts with CEEnv by selecting a sequence of actions, observing subsequent states, and receiving rewards. The episode terminates under two conditions: (1) the RL agent selects an "Idle" action, or (2) the number of action steps reaches the maximum limit, which we set to 100 in this work.

Resource availability in the cloud-edge continuum is inherently dynamic, with computing nodes experiencing fluctuating CPU and memory capacities. To ensure the RL agent learns rescheduling policies under varying resource conditions, we configure CEEnv to randomly assign the CPU and memory availability of each node to 30%, 70%, or 100% during the environment reset. For example, a computing node with a maximum of 16 CPU cores may have its available cores initialized to 4.8 (30% of 16), 11.2 (70% of 16), or 16 (100% of 16). Similarly, a node with a maximum of 8GB memory may have 2.4GB, 5.6GB, or 8GB available. This randomization ensures that at the start of each episode, the RL agent experiences a diverse range of resource availability scenarios.

Following the reset of the cloud-edge continuum, CEEnv generates the MSA application workload based on user-specified configurations. We randomly assign the number of replica pods for each service between 1 and the maximum replica count, which is set to 5 in this study. This approach allows the RL agent to learn rescheduling policies for MSA services with varying degrees of replication, reflecting real-world scenarios where autoscalers adjust workloads to accommodate fluctuating user demands.

After generating the MSA application workloads, the pods are randomly deployed across the simulated cloud-edge computing nodes, establishing the initial placement for the MSA application. The training episode then commences, with the RL agent exploring

[6]https://stable-baselines3.readthedocs.io/en/master/

rescheduling actions to optimize performance under the given resource constraints and application demands.

5.4 Rescheduling Plugin

As discussed in Section 4, current solutions to the MSA application placement problem often focus on the initial scheduling process. In this work, we developed a plugin to enhance the Kubernetes scheduler, enabling it not only to manage the initial scheduling of pods but also to continuously monitor the status of the cloud-edge continuum and the running MSA applications, allowing it to perform rescheduling actions as needed.

Although this work utilizes Reinforcement Learning as the rescheduling policy, the plugin is not specifically tailored for RL agents. Instead, it adheres to the design principles of MSA, where the policy can be independently deployed and integrated with the plugin. This design allows the plugin to be adaptable and compatible with potentially different rescheduling algorithms, making it more flexible for future developments.

Figure 5.5 illustrates the overall architecture of our proposed rescheduling plugin, which consists of three primary components, each designed to address specific aspects of the rescheduling process. The first component, the **Rescheduling-Controller**, functions as the orchestrator, managing the entire rescheduling process. It continuously monitors the MSA application's end-to-end latency and initiates the rescheduling planning process by invoking the **Rescheduling-Planner** component whenever the latency exceeds user-defined thresholds. Upon receiving a rescheduling plan from the Rescheduling-Planner, the Rescheduling-Controller triggers the **Rescheduling-Operator** to execute the required rescheduling actions.

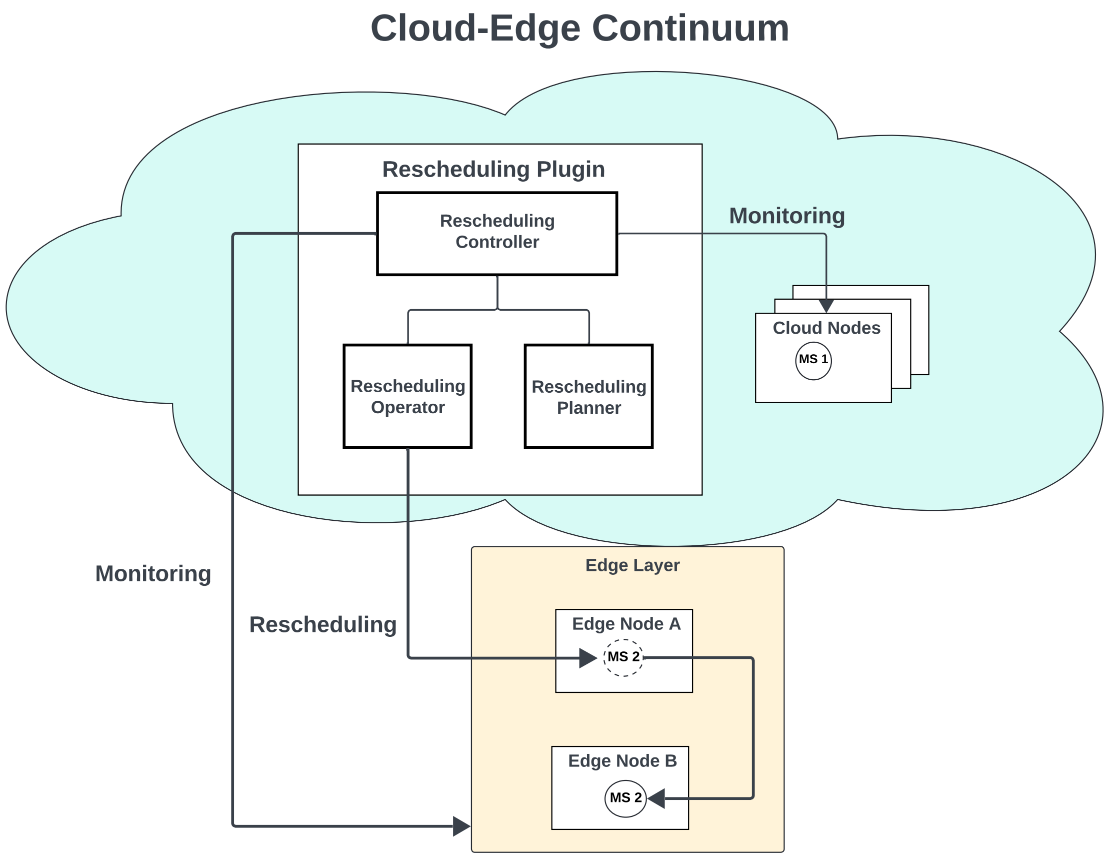


FIGURE 5.5: Overall architecture of the rescheduling plugin, consisting of three main components: the Rescheduling Controller, which continuously monitors application and cluster states; the Rescheduling Planner, which generates rescheduling plans based on observed states; and the Rescheduling Operator, which executes the rescheduling actions.

The second component, the **Rescheduling-Planner**, acts as the primary decision-making engine responsible for generating microservice rescheduling plans. At the core of this component is a rescheduling policy that formulates a rescheduling strategy based on the observed states of cloud and edge computing resources, as well as the operational characteristics of the MSA application. In our work, we employed a reinforcement learning-based rescheduling policy. Initially, the RL agent is trained within the proposed RL simulation environment, CEEnv, as described in Section 5.2. This approach allows for efficient learning without the delays associated with real-world deployments. Once trained, the agent is deployed as the rescheduling policy within the Rescheduling-Planner, enabling real-time rescheduling decisions in a live cloud-edge environment.

The third component, the **Rescheduling-Operator**, is tasked with executing the core rescheduling operations according to the plans generated by the Rescheduling-Planner. This component interacts directly with the cloud-edge continuum to ensure that service instances are efficiently rescheduled to target nodes, minimizing service downtime and ensuring a seamless transition during the rescheduling process.

5.4.1 Rescheduling-Controller

The Rescheduling-Controller is a pivotal component of the proposed Rescheduler, responsible for orchestrating the entire rescheduling process within a control loop. It continuously monitors the application's end-to-end latency by querying the monitoring tools that are deployed in the Kubernetes cluster, it uses the latest latency data to maintain an estimate of the application's end-to-end latency by calculating a moving average latency value from its latest monitored, defined as:

$$\text{Estimated Latency} = (1 - \alpha) \times \text{Estimated Latency} + \alpha \times \text{Sample Latency}$$

where Sample Latency is the most recent latency measurement and α is a smoothing factor set to 0.3 in this study. This moving average approach mitigates the impact of transient latency fluctuations, ensuring that rescheduling decisions are based on stable and reliable data.

When the estimated latency surpasses the user-defined threshold, the Rescheduling Controller initiates the rescheduling process. This process begins by communicating with the container orchestrator to retrieve metrics data, which mainly includes two parts: The first part is the Node Resource Availability Data including the resource availability information of current cloud and edge nodes in the continuum, the second part is the Pod Data, including where each of running pod is hosted and their corresponding resource request for the computing node.

The aforementioned collected data is subsequently passed to the Rescheduling-Planner, which utilizes this information to generate a rescheduling plan. The Rescheduling Controller will then decide whether to invoke the Rescheduling-Operator for actual rescheduling actions based on the different plans returned by the Rescheduling-Planner.

5.4.2 Rescheduling-Planner

The Rescheduling-Planner serves as the core decision-making component, integrating the rescheduling policy and receiving requests from the Rescheduling-Controller. It uses Node Resource Availability Data and Pod Data received from Rescheduling-Controller as input for the underlying rescheduling policy.

The underlying rescheduling policy is responsible for generating two types of plans: a rescheduling action plan and a Skip plan:

Rescheduling Action Plan: This plan specifies a pod to be rescheduled and a target node for pod rescheduling. Formally, the rescheduling action can be represented as $A(i, j)$, where A denotes the action of moving pod i to node j. If a rescheduling action plan is generated, it is forwarded to the Rescheduling-Operator, which executes the actual rescheduling operation.

Skip Plan: This plan indicates that no rescheduling operation will be performed, thereby skipping the current iteration of the control loop. An "Skip" plan is produced by Rescheduling-Planner in the case it thinks that there is no way to improve the end-to-end latency of the MSA application, which will be elaborated in Section 5.4.2.

In this work, we incorporated a step penalty mechanism, discussed in Section 4.3, to train our RL-based rescheduling policy to learn when to generate "Skip" plans. This approach allows the policy to avoid unnecessary rescheduling actions once the placement of pods in the MSA application has converged to an optimal configuration, thereby preventing actions that do not improve the end-to-end latency. Specific strategies were incorporated into the RL agent's training process to achieve this behavior.

Based on the input state of the computing nodes and running pods, the rescheduling policy generates rescheduling plans. One of the critical objectives is for the underlying policy to produce a valid rescheduling plan, ensuring that the subsequent rescheduling process can be successfully executed by the Rescheduling-Controller. In this study, we employed DQN and PPO as our RL-based rescheduling policies. The PPO agent, in particular, is capable of generating valid actions by utilizing the masking mechanism described in Section 4.4.3. Specifically, the PPO agent first creates action masks based on the Node Resource Availability Data and Pod Data, which are then applied during the inference process to ensure that only valid rescheduling actions are selected. In contrast, the DQN agent does not support the use of action masks, which means it may still generate invalid actions that are subsequently handled by the Rescheduling-Controller.

5.4.3 Rescheduling-Operator

The Rescheduling-Operator is the main component for executing the actual rescheduling process to the pods running in the Kubernetes. In this work, we meticulously designed this process to minimize rescheduling overhead, with a primary focus on reducing service downtime. Rescheduling overhead predominantly arises from service interruptions, which can significantly degrade the Quality of Service (QoS) [60]. In scenarios where a service comprises multiple instances, terminating one instance during the rescheduling process can impose additional load on the remaining instances, thereby diminishing QoS. Conversely, for services with only a single instance, the rescheduling process may lead to service interruptions or even data loss, adversely affecting user experience.

To effectively address these challenges, we implement a rescheduling algorithm, which is specifically engineered to eliminate instance downtime. The Rescheduling Operator initiates the rescheduling process by deploying a new service instance on the target node and waits until this new instance attains a running state. Only after the new instance is fully operational is the old instance terminated. This strategy ensures that the total number of replicated service instances remains unchanged throughout the rescheduling process. By maintaining a constant number of active instances, our approach effectively prevents service interruptions and minimizes rescheduling overhead.

5.4.4 Pod rescheduling in Kubernetes

Kubernetes does not natively support pod rescheduling operations. The closest native functionality is provided by the Descheduler [7], which identifies pods that meet specific conditions defined by de-scheduling policies and subsequently terminates them to rebalance pod distribution across nodes. However, once the Descheduler terminates a pod, it is rescheduled by the Kubernetes scheduler without control over the specific node to which it is reassigned. To overcome this limitation, we have developed a custom rescheduling functionality implemented in our rescheduling plugin.

Since the pod is part of a replica set for the service, Kubernetes does not allow specifying the placement of individual pods within the replica set. As a workaround, the Rescheduling-Operator labels all nodes except the target node as “unschedulable”, ensuring that subsequent pod scheduling occurs exclusively on the target node. The Rescheduling-Operator then increases the replica count for the service deployment, prompting Kubernetes to schedule the newly created pod on the target node. After successfully deploying the new pod, the node labels are restored to their original state, making all nodes ready for scheduling again.

Once the new pod is operational on the target node, the Rescheduling-Operator proceeds to remove the old pod. Recall that to schedule the new pod, we increased the replica count of the service deployment; at this stage, the Rescheduling-Operator decreased the replica count. To ensure that the old pod is terminated promptly when the replica count is reduced, the Rescheduling-Operator sets the **Pod Deletion Cost** [8] of the old pod to a low value. The pod deletion cost is a Kubernetes feature that influences the scheduler’s decision on which pods to terminate first when the number of replica pods is decreased, allowing pods with lower deletion costs to be removed first. By assigning a low Pod Deletion Cost to the old pod, we ensure that it is selected for termination when the replica count is decreased. Subsequently, the Rescheduling-Operator decreases the

[7]https://github.com/kubernetes-sigs/descheduler

[8]https://github.com/kubernetes/enhancements/issues/2255

replica count, leading to the termination of the old pod due to its low deletion cost. This approach ensures that the old pod is efficiently removed, completing the rescheduling process with minimal impact on service availability.

5.5 MSA Application Profiler

Modern container orchestration platforms like Kubernetes can easily integrate a variety of monitoring toolsets to observe the operational state of MSA applications. In this research, we utilize a stack of monitoring tools to collect crucial MSA application metrics data. We employ these data in two primary ways. Firstly, the MSA application profiling data, specifically the internal execution time for pods, is fed into our Reinforcement Learning (RL) simulation environment, CEEnv, to simulate the end-to-end latency of MSA applications. Secondly, the proposed rescheduler plugin, which enhances the default Kubernetes scheduler, continuously monitors the real-time end-to-end latency of the running MSA application to trigger rescheduling actions when necessary.

5.5.1 Profiling Data for CEEnv

As discussed in Section 5.2, CEEnv requires capturing the expected execution time for each service running on heterogeneous computing nodes to simulate the end-to-end latency of MSA applications. Therefore, precise profiling of the internal execution time of pods is essential. In this work, we utilize μBench [54] to build and deploy benchmark MSA applications. μBench offers seamless integration with Prometheus [9], an open-source monitoring toolkit that collects and stores its metrics as time-series data. When a pod in μBench receives a user request, it records the start time before executing the computational task and the end time after completing the internal execution. The internal execution time of the pod is then calculated and exposed via its `/metrics` endpoint. Then Prometheus is configured to scrape the metrics data periodically and store it in its database. This approach allows for the precise collection of internal execution times of services, which are extensively utilized by CEEnv.

During the profiling process, we iteratively deploy services within the MSA application to different types of nodes. For each profiling iteration, we utilize the benchmark tools provided by μBench to generate user workloads and subsequently collect the average internal execution time over the duration using **PromQL**, the query language specifically developed for Prometheus, to query the Prometheus server. This query retrieves the

[9]https://prometheus.io/docs/introduction/overview/

average internal execution time for a pod over the specified time interval, as shown in Listing 5.1.

```
sum by (app_name)
(increase(mub_internal_processing_latency_milliseconds_sum{}[1m])) /
sum by (app_name)
(increase(mub_internal_processing_latency_milliseconds_count{}[1m]))
```

Listing 5.1: PromQL to collect data

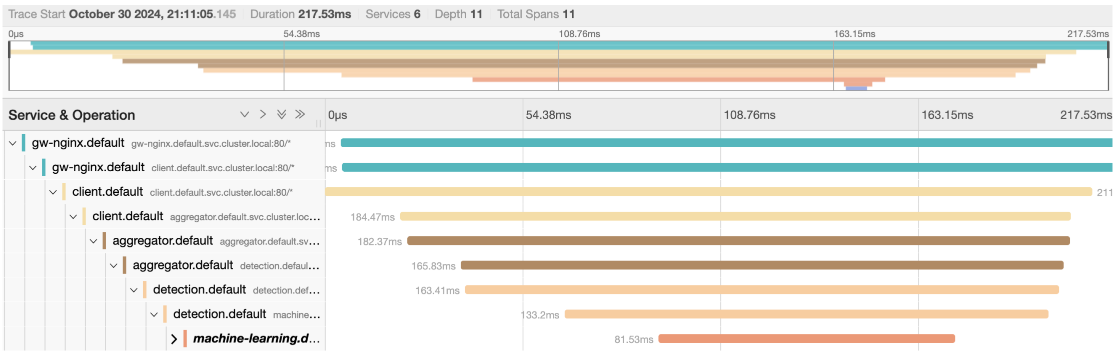
5.5.2 Latency Monitoring for rescheduler

As mentioned in the previous section, the rescheduling controller continuously monitors the end-to-end latency of the MSA application. In this work, we utilize Istio [10], a service mesh technology, to facilitate this monitoring process. Istio allows us to monitor the network traffic of the MSA application in a non-intrusive manner without modifying the application code. This is achieved by injecting a sidecar proxy container into each of the pods in the MSA application. The sidecar proxy intercepts all outbound and inbound traffic of the pods, thereby enabling the capture of service latency.

We considered two approaches to capture the end-to-end MSA application latency. The first approach utilizes the metrics exported by Istio to Prometheus. Istio can be integrated with Prometheus to export service-level latency metrics. However, the data retrieved represents the average latency over a specific time interval (with a minimum granularity of 30 seconds), due to Prometheus's nature of scraping metrics at fixed intervals. Consequently, the metrics collected from Prometheus are not sufficiently responsive to reflect the latest updates in end-to-end latency and are more suitable for time-series analysis.

Therefore, we opted to use Jaeger [11], an open-source distributed tracing system, for more precise latency tracking. Jaeger injects trace headers into requests, which are used to track and record request spans. Figure 5.6 illustrates an example of span data visualized through the Jaeger interface. This span data provides detailed latency information for each microservice involved in a request's invocation path, allowing for more accurate calculation of the end-to-end latency. These spans are collected and aggregated into traces that represent the complete request path through the microservices. Istio integrates seamlessly with Jaeger, enabling automated tracing of microservice interactions. Each proxy sidecar injected into a pod automatically injects the necessary trace headers for Jaeger and sends the collected trace data to Jaeger's backend for analysis.

[10]https://istio.io/latest/
[11]https://www.jaegertracing.io/


Trace Start **October 30 2024, 21:11:05**.145 Duration **217.53ms** Services **6** Depth **11** Total Spans **11**

gw-nginx.default gw-nginx.default.svc.cluster.local:80/* ms
gw-nginx.default client.default.svc.cluster.local:80/* ms
client.default client.default.svc.cluster.local:80/* 211
client.default aggregator.default.svc.cluster.loc… 184.47ms
aggregator.default aggregator.default.sv… 182.37ms
aggregator.default detection.defaul… 165.83ms
detection.default detection.def… 163.41ms
detection.default machine… 133.2ms
machine-learning.d… 81.53ms

FIGURE 5.6: The MSA application span data generated by Jaeger. It includes the execution time of each service along the calling path.

The proposed rescheduler extensively utilizes jaeger to collect microservice latency data. The trace data is collected by the Istio sidecar proxies from each service and then sent to the Jaeger backend. The rescheduling-controller periodically queries the Jaeger backend to retrieve the latest latency data from the MSA application. To smooth out short-term fluctuations and provide a stable metric for decision-making, we are using the latest latency data to calculate a moving average latency for each service as discussed in Section 5.4.1

We calculate the moving average estimated latency for every service that is presented in the trace data, The latency of the entry service of the MSA application is used to estimate the overall end-to-end latency of the entire system. Additionally, the estimated latency of other services in the MSA application is also calculated for a more granular evaluation of the rescheduling effectiveness, enabling a detailed analysis of how rescheduling actions impact the latency of each microservice within the application, which will be used in the evaluation process.

Chapter 6

Performance Evaluation

In this chapter, we present our experimental results, derived from a real-world Kubernetes testbed, to evaluate the performance of proposed PPO-based rescheduling algorithm compared to the baseline algorithms. First, in Section 6.1, we analyze the convergence of the PPO and DQN agents during training. Following this, Sections 6.2, 6.3, and 6.4 describe the baseline algorithms, the configuration of the cloud-edge continuum testbed, and the metrics used in our evaluation. Experimental settings are further detailed in Section 6.5.

In the results analysis presented in Section 6.6.1, we compare the end-to-end latency of benchmark Microservice Architecture (MSA) applications under various scheduling algorithms, observing that the PPO algorithm consistently achieves the most significant latency reduction. In Section 6.6.2, we analyze the execution times of microservices within the MSA applications under each algorithm. This analysis offers insights into how different algorithms impact service execution times, ultimately contributing to the overall end-to-end latency of the applications. Section 6.6.3 explores the distribution of pods across nodes within the cloud-edge continuum, revealing that the PPO-based scheduling policy dynamically adjusts pod placement to accommodate workload fluctuations. Lastly, Section 6.6.4 evaluates the PPO algorithm's resilience to node failure events. Our findings indicate that the PPO algorithm's pod placement strategy reduces average end-to-end latency by **7.8%**, **11.4%**, and **8.8%** for the Chain, Aggregator-Parallel, and Aggregator-Sequential applications, respectively, compared to baseline algorithms. Furthermore, the PPO-based approach consistently outperforms the baselines by effectively minimizing latency variability and suppressing latency spikes.

6.1 Convergence of the RL Agents

In this study, we trained RL agents on a high-performance virtual machine equipped with 32 CPU cores and 64GB of memory, hosted on the Melbourne Research Cloud [1]. For each Microservice Architecture (MSA) application workload, we trained a Proximal Policy Optimization (PPO) agent, which serves as our primary RL model, and compared it against a Deep Q-Network (DQN) agent as a baseline. The hyper-parameters for both PPO and DQN are listed in Table 6.1, it includes both training parameters and reward parameters from our reward function as discussed in Section 4.3. We limited the training time steps to 10 million, resulting in approximately 5 hours of real-world training time for each RL agent.

Parameter	Value	Parameter	Value
$Penalty_{step}$	1.25	Batch size	64
$Penalty_{invalid}$	-10	Replay buffer size	1×10^6
Learning Rate (α)	3×10^{-4}	Exploration Fraction (DQN)	0.1
Discount Factor (γ)	0.99	No. of Fully Connected Layers for Q-Network	3
Clip Range (ϵ)	0.2	No. of Fully Connected Layers for Policy Network	2
Entropy Coefficient (β)	0	No. of Fully Connected Layers for Value Network	2
Soft Update Coefficient (τ)	1		

TABLE 6.1: Hyper-Parameters For DRL Agents

Figure 6.1 shows the accumulated rewards per episodes of PPO and DQN under different MSA workloads, calculated as the average accumulated rewards over every 100 episodes during the training process. We observe that PPO converges faster than DQN. PPO reaches convergence in all workloads at around 2 million time steps, while DQN gradually converges and stabilizes at around 6 million time steps. We believe the difference in convergence rates between DQN and PPO agents mainly stems from the fact that the PPO agent utilizes the invalid action mask as discussed in Section 4.4.3 to ensure its exploration includes only valid actions. This approach spares PPO from learning the valid action policy from scratch.

[1]https://docs.cloud.unimelb.edu.au/

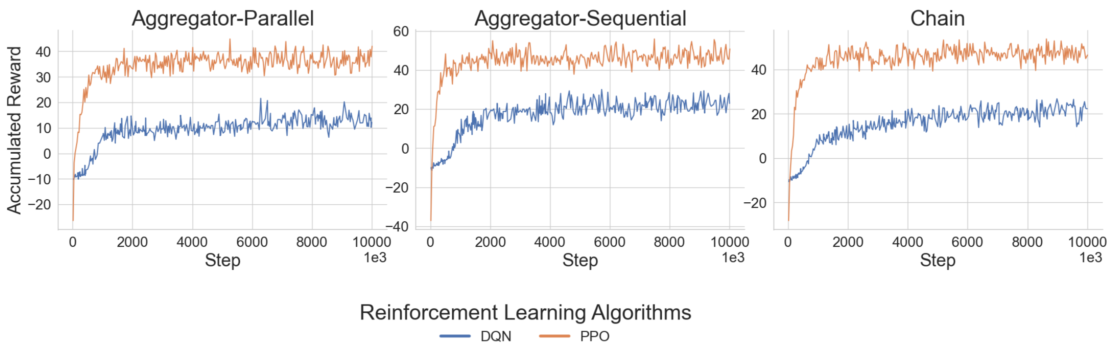


FIGURE 6.1: Convergence of accumulated rewards for PPO and DQN agents on Aggregator-Parallel, Aggregator-Sequential, and Chain application workloads. The PPO agent demonstrates faster convergence in accumulated rewards compared to the DQN agent.

We also observe a significant difference in the converged accumulated rewards achieved by DQN and PPO. This discrepancy is due to the large invalid action penalty applied to DQN. Although DQN learns to have a higher likelihood of generating valid actions, its policy does not fully avoid invalid actions, resulting in a mean average reward significantly lower than that of PPO.

Figure 6.2 shows the average episode length of PPO and DQN. As previously discussed in Section 4.3, an episode ends only if the RL agent chooses the “Idle” action or reaches the maximum episode length, which is 100 in our case. Therefore, the episode length represents the average number of rescheduling actions an RL agent generates before selecting the “Idle” action to actively stop.

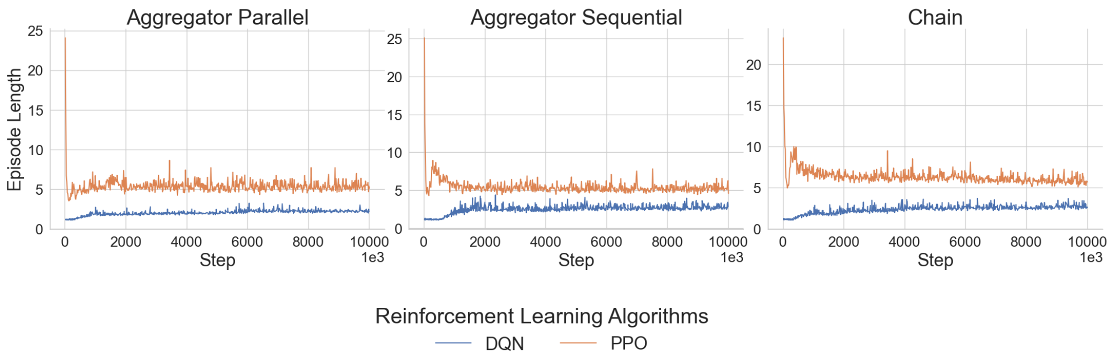


FIGURE 6.2: Average episode length of PPO and DQN agents on Aggregator-Parallel, Aggregator-Sequential, and Chain application workloads. The converged average episode length for PPO exceeds that of DQN, indicating a more explorative policy in performing rescheduling actions.

In the figure, PPO converges to an average of 5 steps per episode, while DQN converges to around 2 steps. The episode length of PPO starts from a high value and then converges, whereas DQN starts from a low value and gradually increases to 2. This difference also arises from how they handle invalid actions. The invalid action penalty makes DQN very conservative in choosing actions other than “Idle”, since invalid actions yield a high penalty. After DQN gradually learns to avoid some invalid actions, it begins to explore actions with a high potential for latency improvement. In contrast, PPO, from the beginning, explores only valid actions due to the masking mechanism, making it more aggressive in pursuing actions that could lead to latency improvements.

6.2 Baseline Algorithms

In this study, we evaluate two types of algorithms: **scheduling** algorithms, which generate initial placements for MSA applications, and **rescheduling** algorithms, which optimize existing placements by performing pod rescheduling. We employ three scheduling algorithms and one rescheduling algorithm as baselines to benchmark the performance of our proposed PPO rescheduling algorithm:

Default: This scheduling algorithm is the default algorithm that is used by the Kubernetes Scheduler [2]. It assigns pods to nodes by ensuring that each pod’s resource requests are satisfied by available nodes. Beyond meeting resource requirements, the scheduler does not impose additional placement constraints, resulting in placements that may vary randomly across different deployments of microservice applications. In a cloud-edge continuum setup, this leads to a balanced utilization of both cloud and edge resources. Since our rescheduling algorithms operate on placements generated by the Kubernetes Scheduler, Default serves as an important baseline to demonstrate the potential latency improvements achievable through rescheduling.

Cloud-First: This scheduling algorithm adopts a best-effort placement strategy that prioritizes offloading jobs to the cloud layers within the cloud-edge continuum. In scenarios where edge computing resources are highly constrained, prioritizing cloud resources helps alleviate resource limitations. Additionally, cloud computing resources generally offer greater computational power and scalability, making them suitable for processing computationally intensive tasks.

Edge-First: In contrast to the Cloud-First strategy, the Edge-First scheduling algorithm prioritizes placing pods on edge-layer devices to leverage the low-latency characteristics inherent to edge computing. This best-effort placement strategy attempts to utilize edge resources first; if edge computing resources are insufficient, pods are then offloaded to cloud layer nodes. This approach aims to minimize latency by keeping computation close to the data source or end-users.

[2] https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

DQN: The DQN rescheduling algorithm utilizes a Deep Q-Learning reinforcement learning approach to derive a rescheduling policy. We use DQN as a baseline to compare against the actor-critic-based reinforcement learning rescheduling algorithm, Proximal Policy Optimization (PPO), to evaluate the benefits of the PPO approach in optimizing pod placements.

6.3 Cluster Setup

6.3.1 Virtual Machine (VM) Setup

We built a real-world cloud-edge continuum testbed using virtual machines (VMs) provided by the Melbourne Research Cloud [3]. The VM specifications are summarized in Table 6.2, with the testbed comprising 7 VMs in total, providing a combined capacity of 32 CPU cores and 128GB of memory. The configuration includes 2 VMs allocated to the cloud layer, 4 VMs forming the edge layer, and 1 VM dedicated to hosting a client pod for generating end-user requests to the MSA applications. Each VM runs Ubuntu 22.04 and is equipped with a 30GB persistent volume. To reflect the heterogeneous nature of computing resources in a typical cloud-edge continuum, we employed three types of VMs with varying computational capacities.

VM Type	CPU Cores	Memory	Storage	Operating System	Count
Cloud-A	8	16GB	30GB	Ubuntu 22.04	2
Edge-A	4	8GB	30GB	Ubuntu 22.04	2
Edge-B	2	4GB	30GB	Ubuntu 22.04	2
Client-A	4	8GB	30GB	Ubuntu 22.04	1

Table 6.2: Testbed Virtual Machines Specifications

We then deployed the Kubernetes container orchestrator platform to manage the testbed clusters. While distribution versions like K3S [4] and K0S [5] offer simplified, one-off installation processes that ease Kubernetes deployment, they are often highly encapsulated and lack the extensibility of the original Kubernetes cluster. Therefore, we chose to manually configure and deploy a native Kubernetes cluster using kubeadm [6], a toolkit designed for deploying Kubernetes on bare-metal clusters. We developed a script to enable crucial Linux system modules on the VMs provided by the Melbourne Research

[3]https://docs.cloud.unimelb.edu.au/
[4]https://k3s.io/
[5]https://k0sproject.io/
[6]https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/

Cloud, which are essential for installing Kubernetes but not well documented by Kubernetes officials. The script is presented in List 6.1.

```
#!/bin/bash
set -e
# Configure persistent loading of modules
tee /etc/modules-load.d/k8s.conf <<EOF
overlay
br_netfilter
EOF
# Ensure load modules
modprobe overlay
modprobe br_netfilter
# Set up required sysctl params
tee /etc/sysctl.d/kubernetes.conf <<EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF
```

LISTING 6.1: Script for enabling linux system module

One of the cloud layer nodes is configured as the Kubernetes control plane. By default, Kubernetes disables the scheduling of workload pods on control-plane nodes to enhance their availability. However, in this study, we enable scheduling on the control plane node since the testbed is intended for non-production use. To facilitate the pod-to-pod networking model inherent to Kubernetes, Calico [7] is deployed as the network plugin. Calico assigns a unique cluster-wide IP address to each pod within the Kubernetes cluster by leveraging its managed IP pools, thereby ensuring efficient and scalable network management. Additionally, Containerd [8] is utilized as the container runtime.

With a functional Kubernetes cluster in place, we proceed to install the monitoring stack detailed in Section 5.5, which comprises Istio, Jaeger, and Prometheus. Prometheus is deployed using the community-maintained Helm chart `kube-prometheus-stack` [9], which also integrates seamlessly with Grafana to enhance data visualization capabilities. For Istio, the official command-line tool `istioctl` [10] is employed to streamline its installation process. Furthermore, a specialized configuration for Zipkin is implemented to enable Istio's integration with Jaeger, as illustrated in Listing 6.2. This comprehensive monitoring setup ensures robust observability and tracing within the cloud-edge continuum testbed.

[7]https://docs.tigera.io/calico/latest/getting-started/kubernetes/self-managed-onprem/onpremises
[8]https://containerd.io/
[9]https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack
[10]https://istio.io/latest/docs/setup/install/istioctl/

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    enableTracing: true
    accessLogFile: /dev/stdout
    defaultConfig:
      tracing:
        zipkin:
          address: zipkin.istio-system:9411 <--- important
```

Listing 6.2: Confuration files for istio to integrate with Jeager

6.3.2 Configurations for Cloud-Edge Continuum Testbed

In this work, we configure the testbed to reflect the network characteristics and heterogeneity of computing resources in the cloud-edge continuum. Cloud and edge nodes in this continuum exhibit varied latencies to end users. We apply latency constraints to two cloud-layer computing nodes using the Linux network traffic control utility `tc` [11]. The raw network latency between virtual machines (VMs) in the Melbourne Research Cloud is under 1ms. For each of the cloud nodes, we introduce a 50ms latency to all edge-layer and user-layer nodes to simulate realistic network conditions.

To emulate the heterogeneity of computing power often found in the cloud-edge continuum, we configure our testbed to include computing nodes with varying performance capabilities, which can lead to differences in execution times for the same tasks. The VMs in the Melbourne Research Cloud offer limited options for CPUs with different per-core performance; therefore, we need to apply constraints to the VMs in the cluster to reflect such computing power heterogeneity.

One approach to achieve this is to limit the CPU frequency using the Linux utility `cpufreq` [12]. However, this requires support for the frequency governor module in the host machine, which is not available in the VMs of the Melbourne Research Cloud. Consequently, we opt to emulate execution time differences directly at the application layer. Each microservice running on cloud nodes of type Cloud-A operates with normal execution time. Services running on edge nodes of type Edge-A are configured to have execution times that are 1.5 times that of Cloud-A, while those on nodes of type Edge-B have execution times set to twice that of Cloud-A.

The extension to application execution time is implemented at the application level: for each service of the MSA application that is constructed by μBench, after completing

[11] https://www.man7.org/linux/man-pages/man8/tc.8.html
[12] https://docs.kernel.org/admin-guide/pm/cpufreq.html

the internal computing tasks, it sleeps for a period based on the node type it is hosted on. This extends the execution time according to our configured factor for each node type. This approach allows us to use the existing cloud computing VM resources from the Melbourne Research Cloud to construct a testbed that reflects heterogeneous nodes with different computing power in the cloud-edge continuum.

6.4 Metrics

In this work, we focus on three key metrics to evaluate the effectiveness of the proposed rescheduling algorithms. The first metric is application *end-to-end latency*, which is the main metric we use to evaluate rescheduling and scheduling algorithms performance. It calculates the time when requests are sent from end users and received by the end users.

The second is the individual *service execution time*, which is the duration starting when a microservice receives a user request until the service response. During the service execution time, it executes internal tasks and calls external services if any. This metric is useful for giving insight into how each service contributes to the MSA application's end-to-end latency. These two metrics are obtained in two forms. The first form is the application span data that is collected using the tracing tool Jaeger, enabling us to precisely monitor and evaluate the response times across different microservices. The second form is the time-series end-to-end latency data that is produced by the workload generator, which will be used to extensively analyze the real-time latency variations and trends when application pods are placed by underlying rescheduling or scheduling algorithms.

In this work, we will also collect pod placement information. By analyzing the distribution of pods across various node types under specific placement policies, we aim to understand how the proposed algorithm assigns workloads in the cloud-edge continuum. Given that the nodes in this continuum exhibit varying network latencies to end-users and different task execution capabilities, the rescheduling algorithm must assign pods to nodes that best match their computational requirements. Additionally, the analysis of pod placement offers insights into the resulting network topology, which is critical in microservices architecture (MSA). Since the services within an MSA application are interdependent, an inefficient network topology can lead to increased data transmission delays between microservices, negatively affecting overall performance. Thus, it is essential to assess whether the rescheduling algorithms produce placements that result in a more efficient network topology, facilitating smoother and faster communication between services.

6.5 Experimental Settings

In this study, we conducted two separate experiments to evaluate the performance of the proposed rescheduling algorithms. The first experiment was a quantitative analysis that focused on measuring the **average end-to-end latency** and **service execution time** of applications across different algorithms. Additionally, we collected data on **pod placement distributions** across various node types, which allowed for a detailed examination of the distinctive characteristics of placement policies employed by both proposed and baseline algorithms. In the second experiment, we design a **node failure scenario** in the Kubernetes testbed and test proposed rescheduling algorithms with baselines in this scenario. Time-series latency data is collected to show the latency trend during the whole workout, which provided valuable insights into how each algorithm's pod placement decisions influence the MSA application's end-to-end latency in real-time, and how they adapt to changes in cluster resource availability.

6.5.1 End-To-End Latency Experiment Settings

In this experiment, we quantitatively assess and compare the end-to-end latency, service execution time and pod distribution outcomes for each algorithm. For each of the algorithms and MSA applications that are tested, the evaluation is conducted using three constructed Microservice Architecture (MSA) applications: Aggregator-Sequential, Aggregator-Parallel, and Chain as discussed in Section 5.1. Since each service within these applications can be replicated across multiple pods, we thoroughly test the algorithms under three different replication settings, where each service has 1, 3, and 5 replicas. This setup results in three distinct pod configurations for each MSA application, comprising 4, 12, and 20 pods respectively. The tested MSA applications with different settings are listed in Table 6.3.

Name	Type	Service Pod #	Total Pod #
chain-4pods	Chain	1	4
chain-12pods	Chain	3	12
chain-20pods	Chain	5	20
aggregator-parallel-4pods	Aggregator-Parallel	1	4 aggregator-parallel-12pods
Aggregator-Parallel	3	12	
aggregator-parallel-20pods	Aggregator-Parallel	5	20
aggregator-sequential-4pods	Aggregator-Sequential	1	4
aggregator-sequential-12pods	Aggregator-Sequential	3	12
aggregator-sequential-20pods	Aggregator-Sequential	5	20

Table 6.3: Benchmark MSA Applications with Different Settings

There are two categories of algorithms evaluated in this experiment: scheduling and rescheduling algorithms. The scheduling algorithms include baseline approaches such as Cloud-First, Edge-First, and Default. On the other hand, the rescheduling algorithms consist of the proposed PPO rescheduling algorithm and the baseline rescheduling algorithm DQN. Since the pod placement happens in different phases of two types of algorithms, quantitative experiment settings are slightly different.

For the scheduling algorithms, the MSA application is deployed directly by the underlying scheduling algorithms. Subsequently, the workload generator Runner [13] provided by μBench is used to generate requests for the MSA applications. We configure the workload generator to operate in "greedy" mode with 1 working thread. In this setting, the workload generator initiates one thread to request the MSA application greedily.

As detailed in Section 5.1, a client pod is placed on specific user-layer nodes within the testbed. The workload generator solely invokes the client pod within the cluster to request the MSA applications, which means the end-to-end latency is measured from the client pod, rather than from the workload generator. In such a setting, our measurement of the end-to-end latency is controlled. After initiating the workload generator, we allow a 60-second stabilization period to ensure that both the workloads and the application deployment reach a steady state. Following this period, the latency metrics and pod distribution data from the running MSA application are collected.

For the rescheduling algorithms, The rescheduling module, which is configured with the underlying rescheduling algorithm is first deployed in the Kubernetes cluster and begins monitoring the cluster and the MSA application state. Subsequently, the MSA application is deployed using the Kubernetes Scheduler, which essentially employs Default scheduling algorithm as discussed in Section 6.2. The workload generator then starts generating workloads with the same settings used for evaluating the scheduling algorithms. Once the MSA application is deployed, the rescheduling module starts to perform a sequence of rescheduling until one of two conditions is met: (1) the rescheduling module produces a "skip" action, or (2) the number of consecutive rescheduling steps reaches the predefined maximum limit, which is set to 15 in this experiment. After the rescheduling process concludes, we wait for 60 seconds and collect the metrics data.

The pod placement information is directly collected from the Kubernetes API server, and the end-to-end latency data is collected from Jaeger. Given the stochastic nature of placement plans generated by the algorithms, it is essential to conduct repetitive experiments. Therefore, we performed 30 repeated experiments for each algorithm and each MSA application tested. This ensures that the results account for variations and

[13]https://github.com/mSvcBench/muBench/tree/main/Benchmarks/Runner

provide a more accurate assessment of algorithm performance. An automated bash script facilitates this process, streamlining data collection and analysis.

6.5.2 Node Failure Experiment Settings

In this experiment, we designed targeted node failure scenarios to evaluate the effectiveness of our proposed rescheduling algorithm compared to baseline algorithms. We conducted experiments individually for each benchmark application—Chain, Aggregator-Sequential, and Aggregator-Parallel—deploying them one at a time onto the testbed. During each application's runtime, a workload generator sent user requests to simulate real-world usage, allowing us to collect end-to-end latency data alongside timestamps. This data enabled us to monitor latency trends and fluctuations over time, providing insight into each algorithm's performance in maintaining or recovering latency under simulated node failure conditions.

In this experiment, each benchmark MSA application is configured with 3 replica pods for each service, totaling 12 running pods per application. We chose this setting because it ensures a sufficiently complex network topology. Additionally, we assume that after a node failure occurs, the remaining computational resources are still adequate to host the underlying MSA applications. MSA applications with 12 pods are recoverable in our node failure scenario, meaning the complete application can still be hosted after the node failures. In contrast, benchmark MSA applications with more pods cannot be fully provisioned after the node failure event.

At the beginning of each experiment, the MSA application is deployed onto the testbed. If the experiment evaluates scheduling algorithms, the initial placement is handled by the underlying scheduling algorithms. Otherwise, in experiments focusing on rescheduling algorithms, pods are initially deployed using Default. After deployment, the workload generator begins sending user requests and collecting end-to-end latency data using the same workload settings discussed in Section 6.5.1. We then wait for 60 seconds to allow the MSA application deployment and workloads to stabilize.

Once the waiting period concludes, the experiment begins. For evaluations involving rescheduling algorithms, rescheduling actions are initiated at this stage. At 150 seconds into the experiment, a node failure event is simulated by failing two nodes simultaneously, representing a significant system disruption. To maintain fairness across algorithms with differing preferences for cloud or edge node placement, we selected one Edge-A node with 4 CPUs and 16GB of memory and one Cloud-A node with 8 CPUs and 32GB of memory. This choice introduces a balanced failure scenario, impacting nodes from both the edge and cloud layers, which allows us to observe each algorithm's adaptability and resilience across the cloud-edge continuum.

To simulate a node failure, we first tag the targeted failed node as "unschedulable," preventing Kubernetes from scheduling pods to it. The Kubernetes API 'evict' [14] is then used to terminate all pods on the failed node. After the node failure event, Kubernetes' self-healing mechanism handles the evicted pods by scheduling them to healthy nodes to maintain the expected number of replica pods for each service. If scheduling algorithms are being evaluated, the self-healing process uses the underlying scheduling algorithm; if rescheduling algorithms are under evaluation, Default is used for self-healing.

After the pods are successfully rescheduled to healthy nodes, rescheduling algorithms (if being evaluated) begin optimizing pod placement. The experiment concludes 150 seconds after the node failure events, and all time-series data is collected for analysis.

6.6 Results

This section presents the end-to-end latency results for benchmark MSA applications across different scheduling algorithms, highlighting that the proposed PPO rescheduling achieves the highest latency reduction in most cases. We proceed by examining the microservice execution times within each MSA application, comparing the outcomes of various algorithms. This analysis provides valuable insights into how each algorithm impacts individual microservice execution times, which collectively determine the overall end-to-end latency. Further, we assess pod distribution patterns under PPO and baseline algorithms, followed by a resilience evaluation of PPO in the face of node failures. In these failure scenarios, the PPO algorithm's pod placement strategy reduces average end-to-end latency by **7.8%**, **11.4%**, and **8.8%** for Chain, Aggregator-Parallel, and Aggregator-Sequential applications, respectively, while also stabilizing latency fluctuations and mitigating performance spikes.

6.6.1 Evaluation of End-to-End Latency

In this subsection, we analyze the end-to-end latency results across the three MSA applications, identifying key performance trends and highlighting the effectiveness of each algorithm in optimizing latency under various configurations.

Figure 6.3 compares the end-to-end latency across various algorithms for three different MSA applications. For each application, three configurations were tested, with pod

[14]https://kubernetes.io/docs/concepts/scheduling-eviction/api-eviction/

counts set to 4, 12, and 20. These configurations were achieved by adjusting the replica count of each service to 1, 3, and 5, respectively.

Overall, the Chain and Aggregator-Sequential applications exhibited higher latency compared to the Aggregator-Parallel. This outcome can be attributed to the shorter maximum calling chain in the Aggregator-Parallel, which benefits from concurrent calls to external services, reducing overall processing time.

Both PPO and DQN rescheduling algorithms start with initial pod placements provided by the Kubernetes Default Scheduler. The proposed PPO rescheduling algorithm outperformed the baselines in most experiments, showing an average reduction of 21.56%, 9.83% in end-to-end latency compared to Default and Second best algorithms in the experiment, DQN. Similarly, DQN demonstrated a 13.01% improvement over Default. These results suggest that both learning based PPO and DQN rescheduling algorithms effectively enhance the initial placement strategy, leading to improved application performance. Across all applications, PPO achieved superior performance over baseline algorithms in the 12 and 20 pod configurations. However, in the 4-pod setup of the Aggregator-Sequential and Aggregator-Parallel applications, DQN delivered better results than PPO.

When comparing DQN to the Cloud-First algorithm, their performance was quite similar. Notably, in the 4-pod setting, the Cloud-First algorithm outperformed all others. However, as the pod count increased to 12 and 20, DQN surpassed Cloud-First. In all configurations, both DQN and Cloud-First showed improvements over the Kubernetes Default Scheduling algorithms.

Despite not being optimized for minimizing end-to-end latency, Default maintained consistent latency across different pod configurations. Compared to heuristic approaches like Cloud-First and Edge-First, the average latency variation from Default across different pod setups remained within 50ms. This consistency indicates that Default can maintain stable latency even as the pod count changes, making it suitable for integration with the Kubernetes autoscaling mechanism.

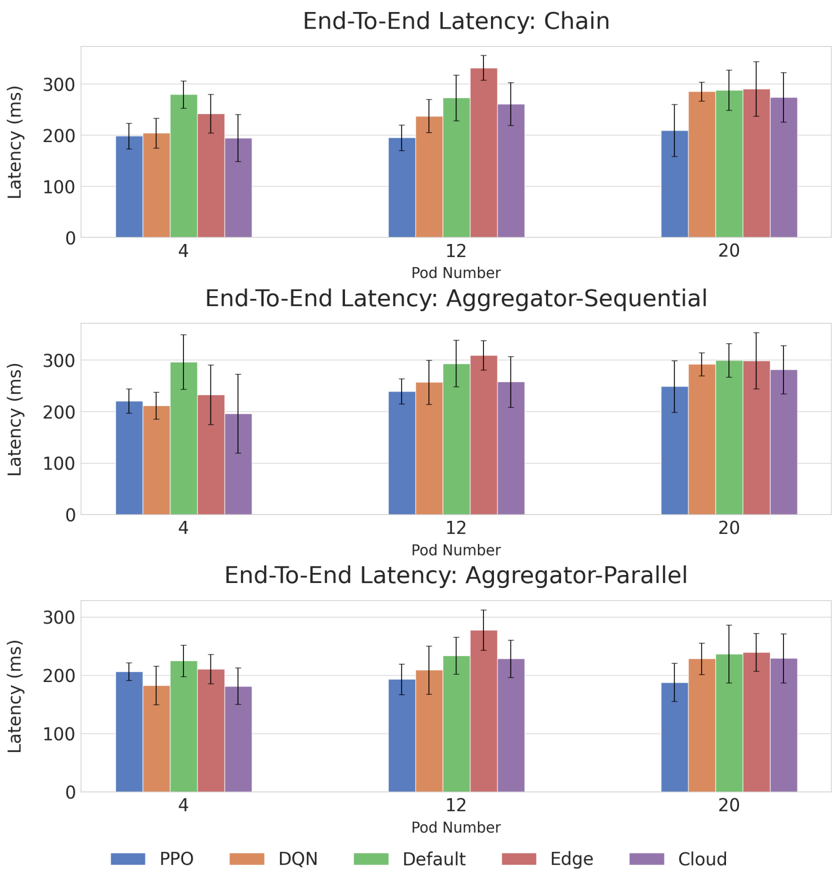


FIGURE 6.3: Comparison of end-to-end latency across three MSA applications. PPO consistently outperforms baseline algorithms in the 12 and 20 pod configurations. In lower pod settings, PPO also closely matches the best heuristic approaches.

The Cloud-First algorithm excelled when the total pod count was low (4 pods), outperforming other algorithms across all applications. This can be explained by its ability to schedule all pods to the cloud layer when the pod count is small. In such cases, internal microservice communication occurs entirely within the cloud layer, resulting in lower network latency. Additionally, although cloud nodes have higher latency to end users compared to edge nodes, they generally provide lower average execution times. Therefore, when the pod count is limited, the cloud's lower processing latency offsets the higher network delay, yielding the lowest overall end-to-end latency.

In our experimental evaluation, Edge-First consistently demonstrated inferior performance when the number of pods increased to 12 and 20 across all application scenarios. This trend was particularly pronounced in Chain, where Edge-First resulted in an overall latency increase of 14.13% compared to Default at 12 pods. Conversely, when the pod count was limited to 4, Edge-First outperformed Default, attributable to its ability to allocate all pods to the edge layer, thereby minimizing inter-microservices network latency.

Our analysis of rescheduling algorithms revealed distinct performance patterns. As the number of pods increased, the Proximal Policy Optimization (PPO) algorithm effectively maintained an average latency of around 200 ms, while the performance of Deep Q-Network (DQN) deteriorated significantly. This difference likely stems from the relative effectiveness of the policies each algorithm has learned. Specifically, PPO exhibited average rescheduling steps of 1.58, 6.2, and 8.95 for configurations with 4, 12, and 20 pods, respectively, indicating a scalable approach to achieving optimal placement as pod numbers grow. In contrast, DQN showed average rescheduling steps of 3.58, 2.47, and 2.09, suggesting inefficiencies with larger pod counts. An insight into this is that DQN frequently produced "skip" rescheduling actions due to the invalid rescheduling actions which selected target nodes lacking sufficient resources, whereas PPO's "skip" actions were valid stop actions facilitated by an invalid mask mechanism, as detailed in Section 4.4. This mechanism ensures that PPO learns policies based solely on valid actions, enhancing its scalability and robustness. Consequently, PPO outperforms DQN in maintaining effective rescheduling policies as pod numbers increase, highlighting its superior adaptability in dynamic environments.

6.6.1.1 Rescheduling Algorithms Overhead

As previously noted, the average rescheduling steps required by the PPO algorithm under 4, 12, and 20 pod configurations were 1.58, 6.2, and 8.95, respectively. Notably, our rescheduling approach selectively adjusts the placement of only 39.5%, 52%, and 42% of active pods on average in these configurations, rather than fully rescheduling the entire application as conventional scheduling algorithms would necessitate. Additionally, as discussed in Section 5.4.4, our algorithm achieves rescheduling by first deploying a new pod on the target node and subsequently terminating the original pod, thereby minimizing disruption. The primary overhead arises from the time required by Kubernetes network plugins to redirect user traffic to the new pod instances. In our tests, this rescheduling process led to an approximately 20% temporary increase in end-to-end latency during the transition phase. However, given the significant latency reduction achieved by the algorithm overall, this overhead remains within an acceptable range.

6.6.2 Service Execution Time

In this section, we examine the service execution times across various MSA applications as influenced by the placement strategies of the PPO algorithm and baseline algorithms. This analysis provides insight into how individual service execution times contribute to the overall end-to-end latency observed with each algorithm.

6.6.2.1 Chain

Figure 6.4 presents the individual service execution times for the Chain application as generated by various scheduling algorithms. As discussed in Section 4.2, service execution time is defined from the moment a service receives a request until it responds, encompassing both internal processing and external service calls. Across all MSA applications evaluated, the Front-End service consistently exhibits the highest execution time due to its role in initiating calls to subsequent services. Specifically, in the Chain application, the Front-End calls the Back-End, which in turn calls the ML service, and finally, the ML service interacts with the DB service. Consequently, the execution latency decreases in the order of Front-End, Back-End, ML, and DB services.

The DB service, performing only lightweight CPU computations and disk I/O operations, contributes minimally to the overall latency, with execution times remaining within milliseconds across different algorithms.

Among the scheduling algorithms, PPO outperforms others in the ML service by maintaining an average execution time below 100 ms across all pod configurations. In contrast, the Edge-First algorithm underperforms in all settings due to the inherently longer execution times of edge computing nodes compared to cloud nodes, which adversely affects the computationally intensive ML tasks. The DQN algorithm demonstrates performance comparable to Cloud-First, while Default does not consistently achieve optimal results. Similar latency patterns observed in the Back-End service reinforce these findings.

For the Front-End service, PPO results in higher execution times than DQN and Cloud-First when the pod count is low (e.g., 4 pods). However, the end-to-end service latency of PPO remains close to Cloud-First and superior to DQN under these conditions. This behavior is attributed to PPO's tendency to allocate the Front-End service to the edge layer while placing other services in the cloud layer. In contrast, DQN and Cloud-First algorithms allocate all pods to the cloud when pod numbers are low, reducing Front-End latency by colocating services within the same layer. Nonetheless, placing services in the cloud introduces additional latency for users, resulting in similar overall performance for PPO, DQN, and Cloud-First when the pod count is limited to four. These results highlight the trade-offs between localized service placement and user-perceived latency, emphasizing the need for adaptive scheduling strategies based on pod configurations.

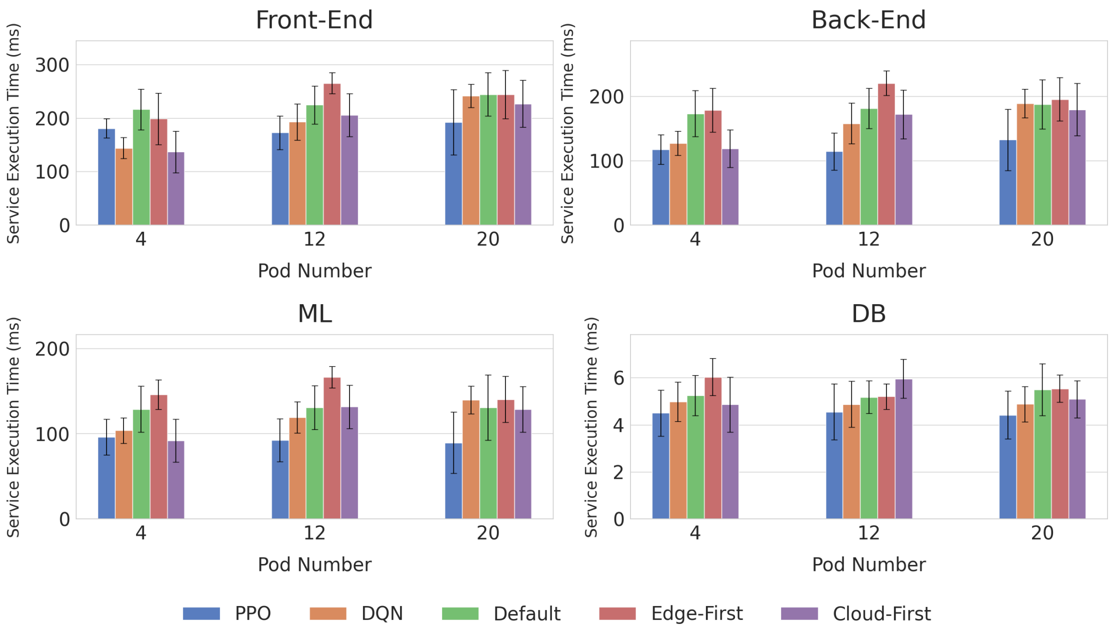


FIGURE 6.4: Execution time of Front-End, Back-End, ML, and DB services in Chain across different algorithms. Each service's execution time contributes to the overall end-to-end application latency.

6.6.2.2 Aggregator-Parallel

The Aggregator-Parallel application involves a Front-End service that concurrently calls the ML and Back-End microservices, with the ML service further interacting with the DB service. Figure 6.5 shows the individual service execution time of Aggregator-Parallel. Compared to the Chain workload, the longest calling chain in the Aggregator-Parallel configuration consists of only three microservices (Front-End, ML, and DB). Consequently, the expected end-to-end latency of the Aggregator-Parallel is generally lower than that of the Chain application.

We analyze the service execution times by tracing from the last services in the calling path, specifically the Back-End and DB services, back to the entry service Front-End. Notably, the Back-End and DB services do not make external service calls, so their execution times primarily reflect internal processing durations. For the Back-End service, PPO produced the best execution times across all pod settings, with DQN's results being close to those of PPO. This indicates that both PPO and DQN effectively rescheduled the Back-End service to computing nodes that offer favorable internal processing performance. Since the DB results were consistent within a few milliseconds, our focus shifted to its calling service, ML.

For the ML service, PPO demonstrated the best performance in the 20-pod configuration, while DQN outperformed the 4 and 12-pod settings. In the 4-pod setup, the

Cloud-First algorithm produced the optimal results. Regarding the Front-End service, Cloud-First also achieved the best performance in the 4-pod setting, while PPO excelled in the 12 and 20-pod configurations. Notably, DQN showed better results than PPO in the 4-pod scenario.

An important observation from these results is that, although PPO did not consistently achieve the best execution times for the ML service in the 4 and 12-pod configurations, it still managed to deliver competitive results for the Front-End service and overall end-to-end latency. This can be explained by the structure of the Aggregator-Parallel application: the external service time of the Front-End encompasses concurrent calls to both the ML and Back-End services. Thus, PPO was able to achieve a lower overall end-to-end latency by strategically balancing the latencies between calls to the Back-End and ML services, rather than optimizing a single service's performance in isolation.

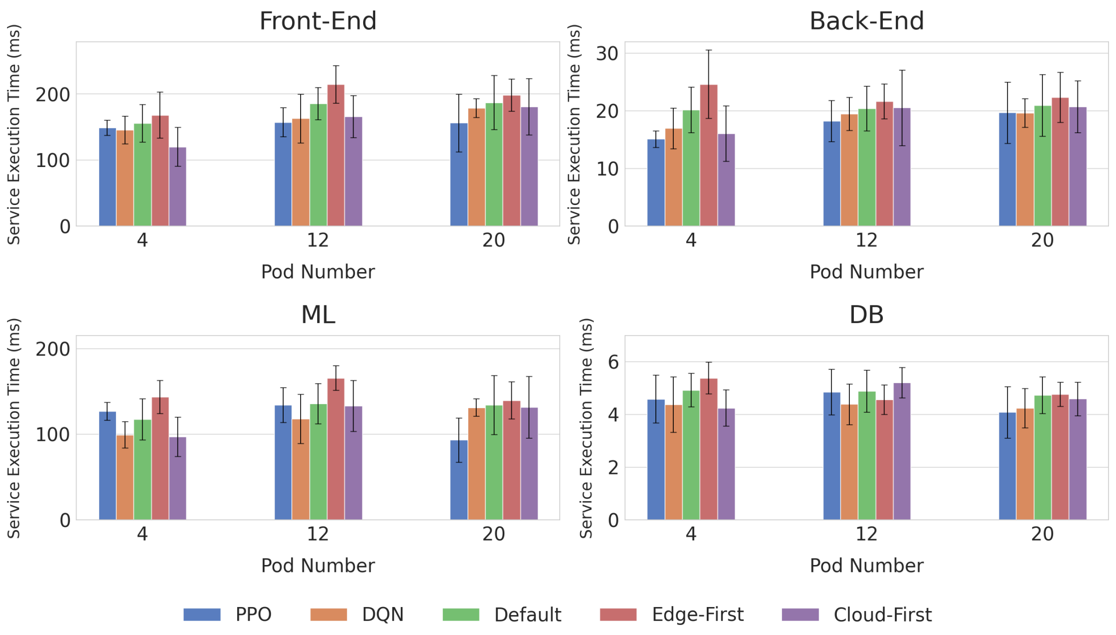


FIGURE 6.5: Execution time of Front-End, Back-End, ML, and DB services in Aggregator-Parallel across different algorithms. Each service's execution time contributes to the overall end-to-end application latency.

6.6.2.3 Aggregator-Sequential

Figure 6.6 shows the individual service execution time of Aggregator-Sequential. In this application, the Front-End service sequentially calls the ML, Back-End, and DB services. Each of these three services does not make external calls, meaning their execution times solely reflect internal processing.

For the Front-End service, the PPO algorithm achieved the best results when the pod count was set to 12 and 20, while DQN outperformed other algorithms in the 4-pod configuration. The Cloud-First approach showed optimal results when the Front-End and all associated services were placed entirely on the cloud layer, taking advantage of lower internal service execution times.

An important observation from the results is that DQN generally outperformed PPO across almost all pod settings for the ML, Back-End, and DB services. However, PPO still achieved superior overall performance for the Front-End service and in terms of end-to-end latency. Analysis of the pod placements generated by PPO reveals that, for the Aggregator-Sequential application, PPO tends to keeps pods on the cloud layer and only moves the ML service to the cloud layer when there is no remaining capacity at the edge. Conversely, DQN is more inclined to place pods predominantly in the cloud layer, regardless of edge capacity.

The key to this pattern lies in the execution and network trade-offs: cloud nodes typically offer lower internal service execution times, which benefits ML, Back-End, and DB services, leading to DQN's better performance for these individual services. However, the network latency between the end user and the cloud nodes can be higher. PPO's approach, which strategically distributes pods, taking advantage of both cloud and edge resources, allows it to achieve a balanced reduction in latency. This results in better overall performance for the Front-End service and lower end-to-end latency, especially when the pod count is 12 or 20, where the network delay becomes a more significant factor.

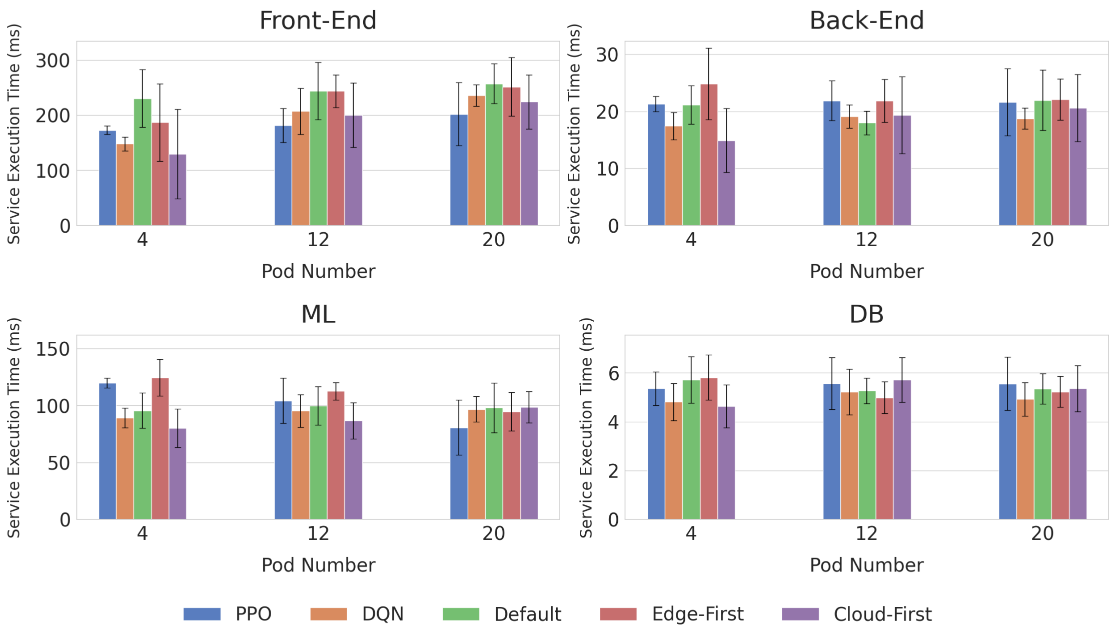


FIGURE 6.6: Execution time of Front-End, Back-End, ML, and DB services in Aggregator-Sequential across different algorithms. Each service's execution time contributes to the overall end-to-end application latency.

By comparing the performance of various algorithms on the end-to-end latency across three MSA applications, the PPO rescheduling algorithm consistently demonstrated superior results compared to both heuristic-based approaches and the DQN rescheduling algorithm in most configurations. Analyzing the execution times of individual services provided deeper insights into how each service contributes to the overall end-to-end latency. Specifically, the optimization strategies employed by PPO effectively reduced the execution time of critical services, thereby minimizing delays along the service invocation paths.

6.6.3 Evaluation of Pod Distributions

In this section, we analyze the pod placement data by generating pod distributions for each service's pods scheduled by different algorithms across various types of nodes. This analysis provides insights into how each rescheduling and scheduling algorithm distributes pods from different services over distinct node types. We evaluated the results separately for three MSA applications, utilizing the maximum 20-pod configuration to reveal more intricate distribution patterns.

6.6.3.1 Chain

For the Chain application, Figure 6.7 presents four subgraphs, each corresponding to a microservice. Each subgraph contains bar charts representing the percentage of pods distributed on different type of computing nodes, where the x-axis indicates node types (Edge-A, Edge-B, Cloud-A) with different computational capabilities as discussed in Section 4.1.2, and the y-axis shows the percentage of pod distribution (ranging from 0 to 1). These distributions illustrate how different algorithms contribute to scheduling pods across various nodes.

For the Front-End service, the PPO algorithm exclusively places pods on Edge-A and Edge-A nodes, with a majority located on Edge-A. Notably, PPO does not schedule any Front-End pods on cloud nodes throughout the 100 repeated experiments. DQN also places most Front-End pods on edge nodes but occasionally schedules them on the cloud. In contrast, other scheduling algorithms display more balanced distributions: Cloud-First allocates more pods to cloud nodes, while Edge-First assigns a greater proportion to edge nodes.

Regarding the Back-End service, both PPO and DQN distribute pods across all three node types, though with distinct patterns. PPO tends to place more Back-End pods on cloud nodes and often allocates them to Edge-A, whereas DQN prefers edge nodes more.

As the Back-End service has a moderate workload, placing it on Edge-A or Edge-A nodes can reduce end-to-end latency due to the availability of more computing resources. The Cloud-First algorithm predominantly schedules Back-End pods to cloud nodes, with significantly fewer placed on edge nodes. In contrast, Edge-First and Default distribute Back-End pods more evenly across all node types.

A critical observation for the ML service is that PPO exclusively schedules ML pods on cloud nodes, setting it apart from other algorithms. This placement strategy has proven to be an effective optimization, as discussed later. DQN, however, distributes ML pods across all three node types, with a notable preference for Edge-A nodes. For the DB service, PPO similarly learns to place all DB pods on cloud nodes, ensuring high computational efficiency. DQN, by comparison, frequently places DB pods on edge nodes. Other scheduling algorithms show a more distributed approach, spreading DB pods across all node types.

These results highlight distinct scheduling behaviors. PPO's strategy of concentrating ML and DB services on cloud nodes reflects a targeted approach to reduce execution times for resource-intensive tasks while balancing Front-End and Back-End placements across edge nodes to optimize overall latency. DQN's more dispersed pod placement may lead to varied performance outcomes, as it does not consistently capitalize on the computational advantages of specific node types.

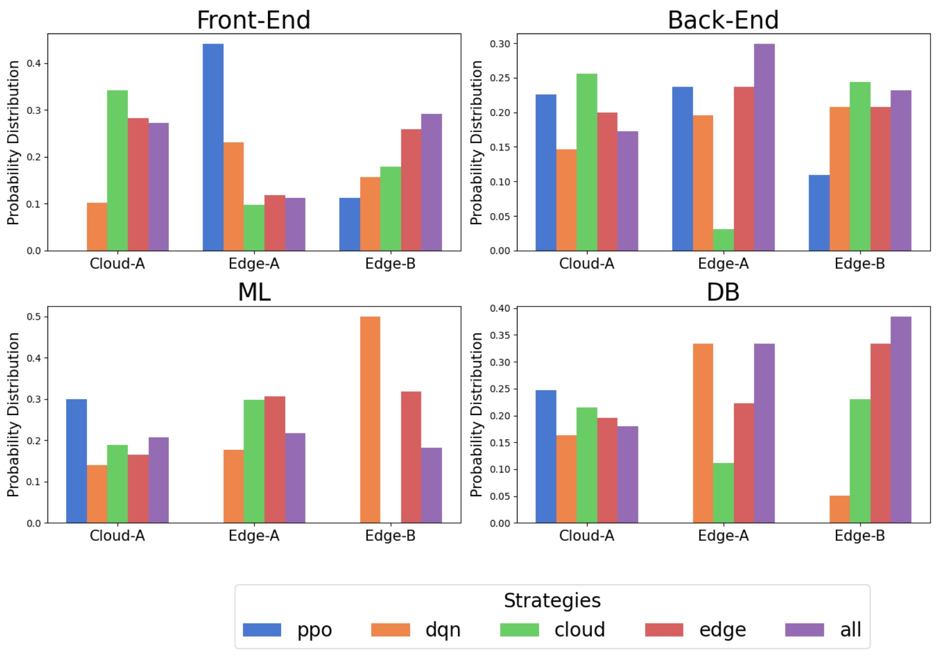


FIGURE 6.7: Pod distribution of Chain application services placed by different algorithms across three node types (Cloud-A, Edge-A, Edge-B). The proposed PPO algorithm tends to schedule the Front-End on Edge-A nodes, ML and DB services on Cloud-A nodes, and distributes the Back-End service more evenly across all node types.

The placement pattern observed from these results shows why PPO generates the overall best end-to-end application latency. Figure 6.8 shows an abstract placement resulting from the rescheduling of PPO and alone with an invocation pattern of Chain application, where each circle represents that the pods from specific services are deployed in the node. In the resulting placement, ML and DB are only placed on the cloud layer, while Front-End is only placed on the edge layer. Even though DB is a lightweight task, PPO learns to schedule it alone with ML to the cloud layers so that the request from ML is not forwarded to the edge layer, which could lead to extra latency. As shown in the Figure, the Chain application only has one calling path, by utilizing such placement, all the requests from the users will only need to go through between the cloud and edge layer once, which incurs minimum layer-to-layer latency.

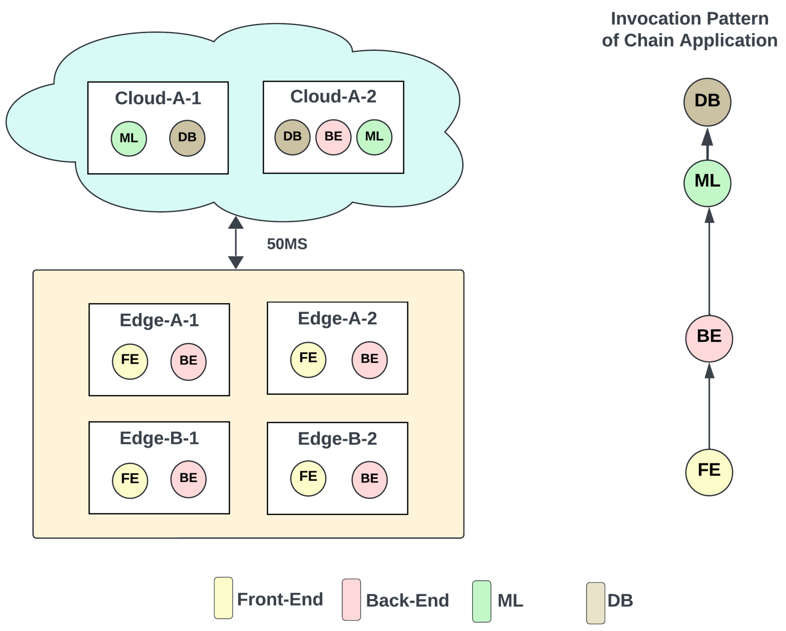


FIGURE 6.8: Chain application placement generated by PPO. The right side shows the Chain application's invocation pattern for reference. The PPO policy tends to place the heavy-lifting ML task and its related DB services on cloud nodes, while keeping lightweight tasks on the edge layer.

6.6.3.2 Aggregator-Parallel

Figure 6.9 illustrates the pod placement distribution for the Aggregator-Parallel application. In this scenario, the Cloud-First, Default, and Edge-First algorithms produced similar results across different tested MSA applications, as these applications only differ in their invocation patterns. Therefore, the discussion will primarily focus on the pod distribution strategies of the PPO and DQN algorithms.

Unlike the Chain application, where PPO predominantly placed Front-End services on edge nodes, for the Aggregator-Parallel application, PPO now allocates Front-End pods to cloud nodes. Both PPO and DQN distribute Front-End pods across all three types of nodes, with PPO showing a preference for Edge-A nodes, while DQN tends to allocate more to Edge-A nodes.

For the Back-End service, both algorithms continue to place pods across all node types. However, PPO generally assigns more Back-End pods to the edge layer, while DQN favors cloud nodes. The distribution patterns for the ML and DB services are consistent with those observed in the Chain application: PPO places almost all ML and DB pods on cloud nodes, whereas DQN distributes ML pods across all three node types.

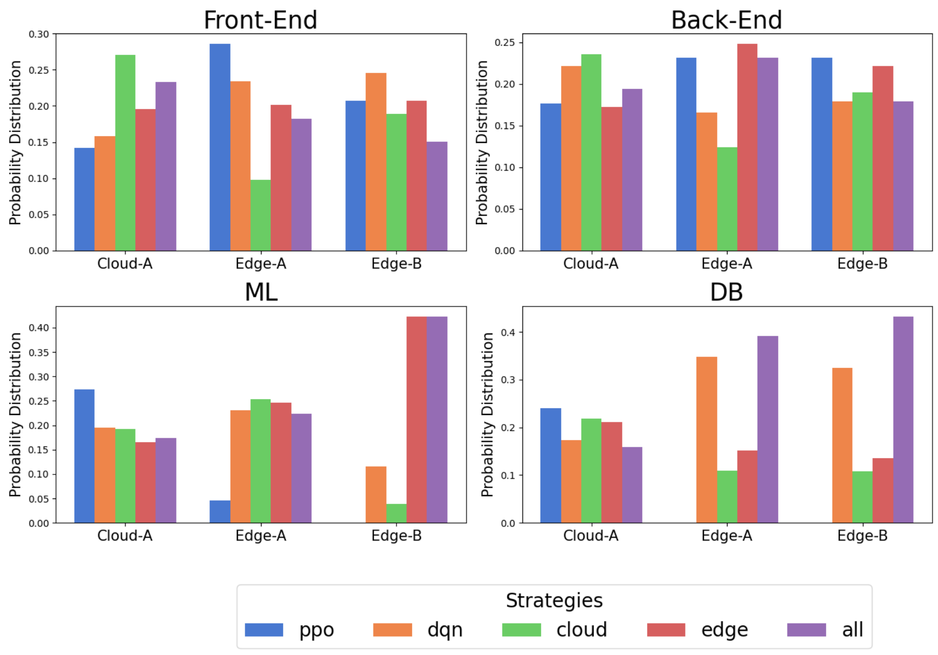


FIGURE 6.9: Pod distribution of Chain application services placed by different algorithms across three node types (Cloud-A, Edge-A, Edge-B). Compared to baseline methods, PPO places all ML and DB services on Cloud-A nodes, while distributing the Front-End and Back-End services across edge nodes.

The end-to-end latency results from Section 6.6.1 confirm that PPO provides the most effective scheduling policy for the underlying workloads. Figure 6.10 depicts a pod placement generated by PPO. Compared to the Chain application, PPO's strategy of placing both Front-End and Back-End pods on the cloud layer still achieves relatively low end-to-end latency. This is mainly because, in the Aggregator-Parallel setup, the Front-End service performs concurrent calls to both the ML and Back-End services, leading the overall latency to be determined by the longest of these concurrent paths. The potential worst-case scenario, where a cloud-hosted Front-End service calls a Back-End service located on the edge layer, is mitigated by the fact that this call occurs concurrently with a call to the ML service. Thus, the overall external service calling time for the front-end is not significantly impacted.

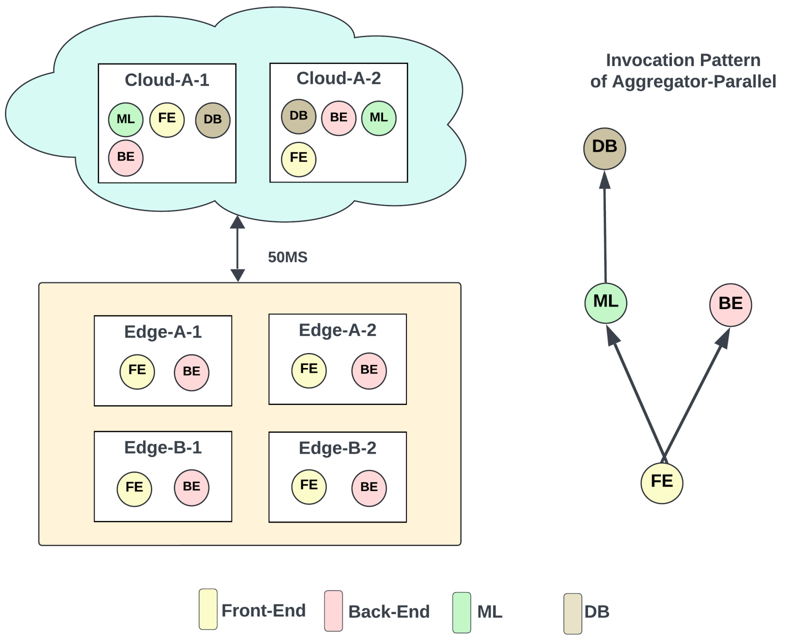


Figure 6.10: Aggregator-Parallel application placement generated by PPO. The right side shows the application's invocation pattern for reference. Similar to the Chain application pattern, the PPO policy places the heavy-lifting ML task and its associated DB service on cloud nodes, while keeping lightweight tasks on the edge layer.

In the Aggregator-Parallel Application, The main advantage of PPO's placement policy over other algorithms lies in its consistent co-location of ML and DB services on the cloud layer. This minimizes latency along the ML service's invocation path, ensuring that critical processing tasks have the lowest possible delays. Consequently, this optimization underpins PPO's ability to deliver superior end-to-end latency compared to other scheduling algorithms, effectively leveraging cloud resources to streamline computationally intensive operations.

6.6.3.3 Aggregator-Sequential

Figure 6.11 presents the pod distribution results for the Aggregator-Sequential application. Similar to the Chain application, PPO places the majority of Front-End service pods on the edge layer, showing a preference for Edge-A nodes. In contrast, DQN distributes Front-End pods more evenly across different node types. For the Back-End service, PPO continues to prioritize placement on edge nodes. However, for the ML service, PPO shifts its strategy and begins to prioritize cloud nodes, while DQN prefers to schedule ML pods on both Edge-A and Edge-A nodes. For the DB service, PPO predominantly schedules pods to the edge layer.

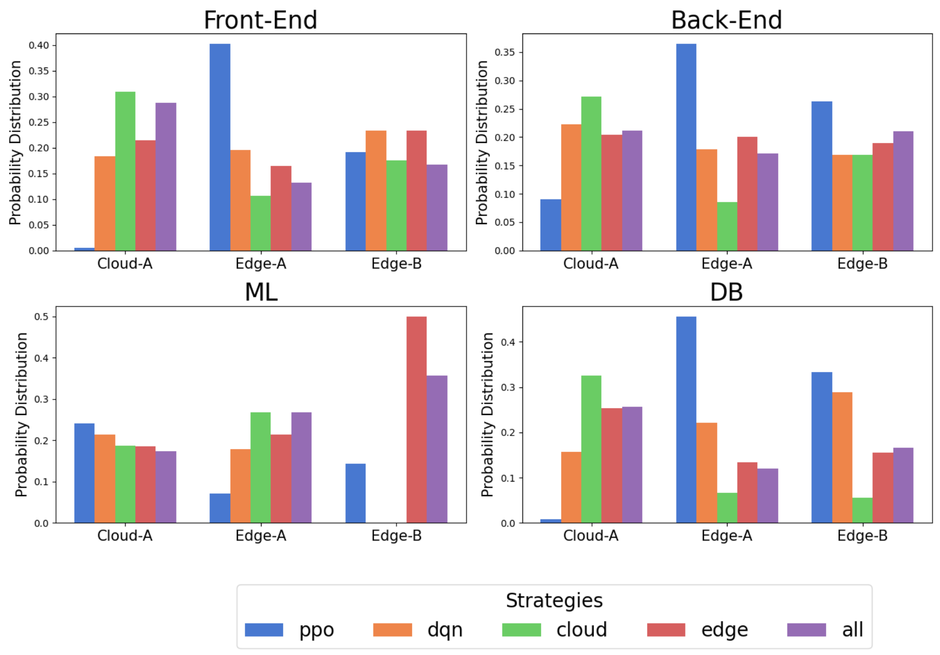


FIGURE 6.11: Pod distribution of Aggregator-Sequential application services placed by different algorithms across three node types (Cloud-A, Edge-A, Edge-B). Compared to baseline methods, PPO places Front-End, Back-End, and DB primarily on edge nodes, while assigning the computationally intensive ML tasks to cloud nodes.

In the Aggregator-Sequential application, PPO has learned to prioritize placing computationally intensive microservices, such as ML, on cloud nodes, while maintaining most other service pods on the edge layer. Figure 6.12 illustrates a typical placement outcome generated by PPO, where only the ML and Back-End services are positioned on the cloud layer, while all other services are retained on edge nodes.

This strategic placement is crucial because, in the Aggregator-Sequential setup, the Front-End service sequentially invokes other services, aggregating their responses to produce the final result. The overall execution time of the Front-End service is the cumulative latency of each service it calls. By offloading the ML service to the cloud, PPO ensures that the most resource-intensive and computationally demanding microservice is hosted on nodes capable of handling such workloads efficiently. This approach also ensures that sufficient resources remain available on the edge layer to provision other services, benefiting from the naturally lower latency of edge nodes. Consequently, this configuration helps optimize the overall latency of the application by balancing computational demands with network efficiency.

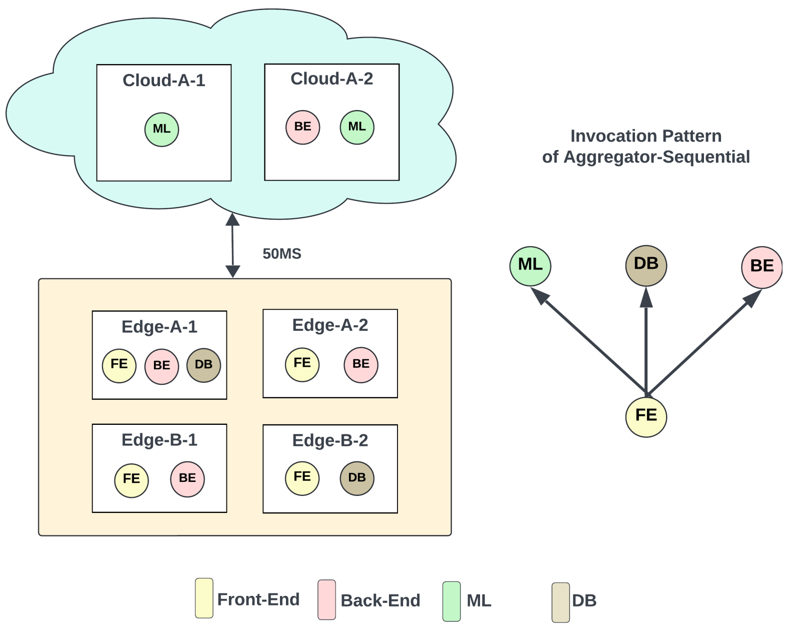


FIGURE 6.12: Aggregator-Sequential application placement generated by PPO. The right side shows the application's invocation pattern. In this case, as the Front-End server sequentially calls ML, DB, and Back-End, PPO makes a best effort to place all tasks except ML on edge nodes, yielding significant results in reducing end-to-end latency.

6.6.4 Evaluation of Adaptability to Cluster Dynamic Changes

In this experiment, we designed a node failure scenario to evaluate the adaptability of proposed rescheduling algorithm to cluster dynamic changes. Each experiment starts by deploying the MSA application on the testbed alongside the respective scheduling or rescheduling algorithms. A simulated node failure event is triggered at a specific time, resulting in the failure of one Edge-A type and one Cloud-A type node in the testbed. Consequently, all pods from the MSA application on these failed nodes are scheduled to the remaining available nodes through Kubernetes' self-healing mechanisms. As described in Section 6.5.2, the rescheduling algorithms, PPO and DQN, work in conjunction with the Kubernetes Default Scheduler to optimize pod placement after the initial deployment. By contrast, the heuristic scheduling algorithms directly manage pod placement during the initial scheduling process.

This design of this experimental setup allows us to examine each algorithm's ability to adapt to dynamic changes in resource availability and its effectiveness in maintaining latency stability during node failures. Throughout the experiment, time-series data on end-to-end latency is collected, providing insights into each algorithm's capability not only to minimize application latency but also to control latency fluctuations and spikes under changing conditions.

6.6.4.1 Chain

Figure 6.13 illustrates the end-to-end latency of the Chain application over time, resulting from different underlying scheduling or rescheduling algorithms. Each subgraph corresponds to a specific algorithm and displays the application's latency variations throughout the experiment. Due to significant fluctuations in latency within the cloud-edge continuum environment, we also include a moving average line in each subgraph to better represent overall latency trends. The moving average smooths out short-term variations by averaging the past data points, providing a clearer view of latency patterns over time. In this research, we set the number of averaging data points to 30.

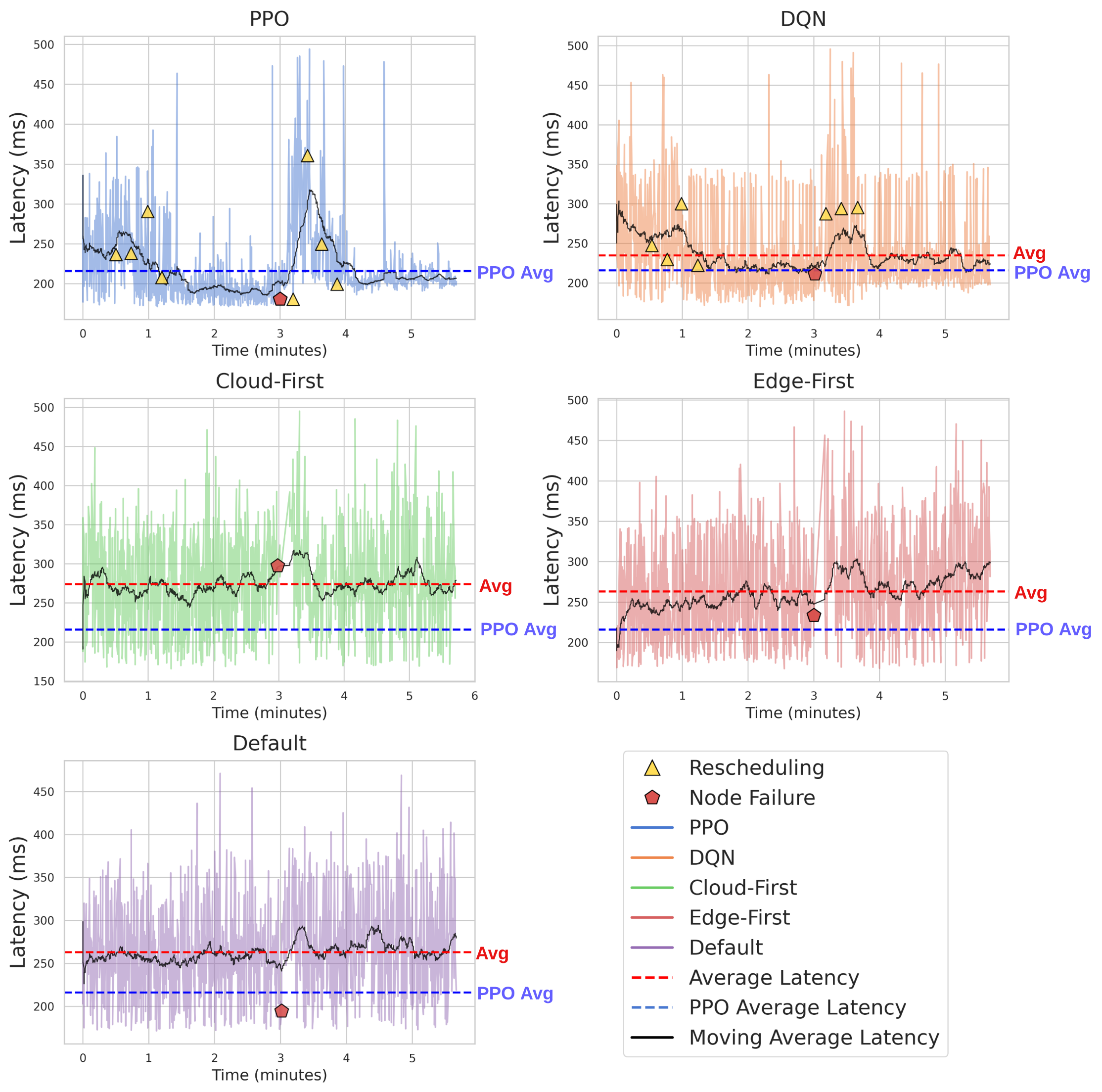


FIGURE 6.13: Latency trends for Chain across algorithms. Rescheduling events are marked with yellow triangles in PPO and DQN, and node failures are marked with orange pentagons across all algorithms. Each sub-graph shows PPO's latency as a blue dashed line, comparing its average latency to each algorithm's average (red dashed line). PPO outperforming all algorithms overall, effectively reducing fluctuations and eliminating spikes.

In each subgraph, the orange pentagon markers represent node failure events. In the subgraphs for the rescheduling algorithms PPO and DQN, yellow triangle markers indicate pod rescheduling events initiated by these algorithms. Additionally, each subgraph includes a black dashed line showcasing the average latency over the entire experiment.

Overall, PPO shows superior results in maintaining low latencies, where 85% of requests are maintained under 250 ms, compared to 76.9%, 35.5%, 46.2%, and 46.7% as achieved by the DQN, Cloud-First , Edge-First, and Default algorithms. Our first observation from the results is that the latency of MSA applications under the heuristic baselines exhibits significant fluctuations, with numerous latency spikes. As discussed in Section 6.6.3, the placement of application pods within a cloud-edge environment greatly impacts end-to-end latency. Given that each service in our MSA application can include multiple replica pods distributed across different nodes, the end-to-end latency varies considerably depending on the specific pods to which user requests are routed. This variability implies that suboptimal pod placement within any service can lead to substantial latency fluctuations.

For the heuristic scheduling algorithms—Cloud-First, Edge-First, and Default—the latency consistently fluctuates between 200 ms and 350 ms, with numerous spikes reaching up to 500 ms. Among all heuristic baselines, Default has fewer requests exceeding 400 ms in latency, accounting for 12.21% of the total requests, whereas the other two heuristic baselines Cloud-First and Edge-First have a higher proportion of high-latency requests (over 400 ms), accounting for 16.73% and 15.37% of the total requests. After the node failure events, the overall latency resulting from the Edge-First pod placement slightly increases, whereas the other heuristic algorithms maintain a similar level of end-to-end latency.

Compared to the heuristic baselines, the latency produced by the DQN rescheduling algorithm also exhibits fluctuations; however, following rescheduling actions, the number of requests with latency below 250 ms rises significantly, which is 76.9% compared to the 35.5%, 46.2%, and 46.7% of the Cloud-First, Edge-First, and Default algorithm, shows its improvement over the heuristics algorithms. This adjustment yields an average end-to-end latency of 234.41 ms for DQN, notably lower than the baseline latencies of 274.11 ms, 263.28 ms, and 263.05 ms, respectively. In response to a node failure, DQN performs three pods rescheduling to optimize pod placement. This rescheduling leads to a more stable end-to-end latency compared to the heuristic baselines' pod placement after node failure, with less fluctuation. In response to a node failure, DQN initiates three pod rescheduling actions to optimize pod placement. This results in slightly increased overall latency compared to the time before the failure and a relatively stable latency curve, though fluctuations remain evident.

The PPO strategy demonstrates superior performance in reducing latency fluctuations for MSA applications. Compared to other strategies, PPO maintains a significantly more stable latency curve with fewer spikes after rescheduling actions. During the whole experiment process, PPO maintains 86.9% of the request latencies under 250 ms. Before the node failures, all request latencies are kept under 300 ms, with most concentrated under 200 ms. After the node failure, the end-to-end latency initially spikes, but after four rescheduling steps performed by PPO, the overall latency is again controlled under 300 ms, with much less fluctuation compared to others. Although the overall end-to-end latency after the node failure is slightly higher than before, it remains around 200 ms. The average end-to-end latency of PPO is 215.98 ms, which significantly outperforms the other baselines.

6.6.4.2 Aggregator-Parallel

Figure 6.14 displays the end-to-end latency of the Aggregator-Parallel application over time. Similar to observations from the Chain application, the latency fluctuations for heuristic-based scheduling algorithms remain significant, with application latency consistently fluctuating between 150 ms and 300 ms. Regarding the average latency over the experiment, the Cloud-First algorithm performs better, achieving a lower average latency of 210.40 ms compared to 219.63 ms and 225.76 ms for the Edge-First and Default, respectively. After the node failures occurred, the overall latency for Cloud-First and Edge-First increased, while the overall latency for Default slightly decreased. A possible explanation is that the node failures concentrated pods onto fewer computing nodes. Since Default initially deploys pods sparsely across nodes, this unintended concentration could reduce internal communication delays within the application, leading to a lower overall end-to-end latency.

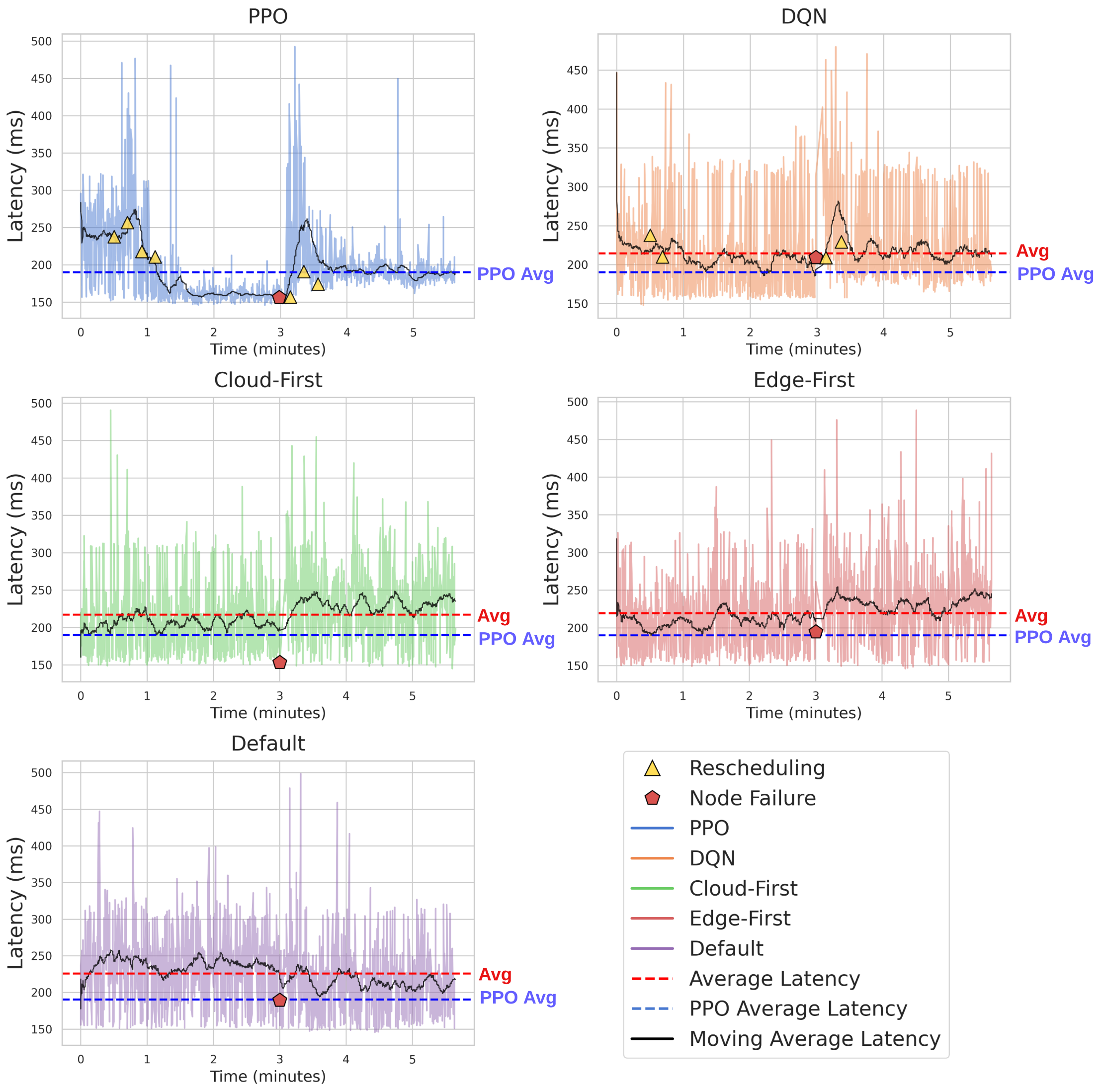


FIGURE 6.14: Latency trends for Aggregator-Parallel across algorithms. In the rescheduling algorithms PPO and DQN, rescheduling events are marked with yellow triangles, and node failures are marked with orange pentagons across all algorithms. Each sub-graph shows PPO's latency as a blue dashed line, comparing its average latency to each algorithm's average (red dashed line). PPO outperforming all algorithms, effectively reducing fluctuations and eliminating spikes.

For the rescheduling-based algorithms, the latency trend pattern of DQN is similar to that observed with the Chain application. Overall, the request latency fluctuates but is slightly better than that of the heuristic scheduling algorithms, with more request latencies falling under 250 ms. DQN performs two pod rescheduling actions before and after the node failures, which is slightly fewer than in the Chain application, where four and three rescheduling actions were issued, respectively. Although DQN slightly reduces the application's latency fluctuations, its average end-to-end latency of 214.63 ms is slightly worse than that of the Cloud-First approach (210.40 ms) but better than the other heuristic baselines.

Compared to all other algorithms, PPO once again yields the best results in controlling latency fluctuations, eliminating latency spikes, and lowering the overall application end-to-end latency. After performing four pod reschedulings before the node failures, PPO decreases the end-to-end latency to around 160 ms, maintaining fluctuations within 10 ms. During this period, the pod placement resulting from PPO's rescheduling eliminates latency spikes, with the highest recorded latency being 207 ms. After the node failures, PPO issues three rescheduling actions and maintains a latency of around 190 ms. Although the overall latency and its fluctuations slightly increase, and a latency spike is observed, this is a reasonable outcome given the reduced availability of computing resources due to node failures. Importantly, PPO still outperforms all baseline algorithms even after the node failures. The final average latency of PPO is 190.25 ms, which is 10% less than that of the best baseline algorithm, Cloud-First.

6.6.4.3 Aggregator-Sequential

Figure 6.15 presents the results for the Aggregator-Sequential application. In this MSA application, the results of the heuristic baselines are close, with average latencies during the experiment of 264.43 ms for Cloud-First, 265.54 ms for Edge-First, and 265.22 ms for Default. The fluctuation of the heuristics-based algorithms is substantial, and the latency spikes are severe. After the node failures, all heuristic scheduling algorithms produced even more latency spikes. Observed from the moving average lines, the overall latency of Default increased, while the others remained essentially the same.

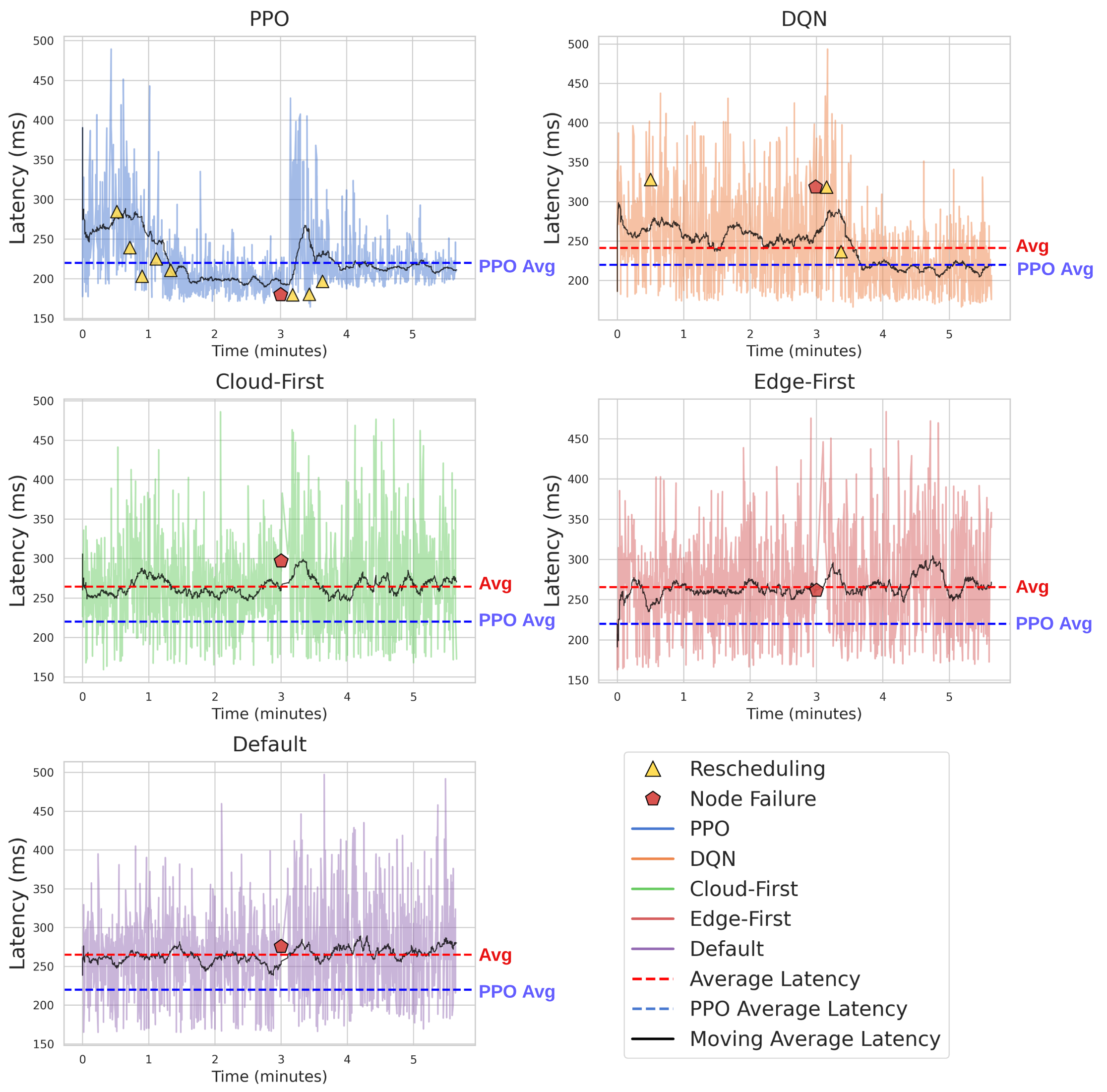


FIGURE 6.15: Latency trends for Aggregator-Sequential across algorithms. In the rescheduling algorithms PPO and DQN, rescheduling events are marked with yellow triangles, and node failures are marked with orange pentagons across all algorithms. Each sub-graph shows PPO's latency as a blue dashed line, comparing its average latency to each algorithm's average (red dashed line). PPO outperforming all algorithms, effectively reducing fluctuations and eliminating spikes.

The performance of DQN in Aggregator-Sequential is noteworthy. Before the node failures, the DQN agent performs only one rescheduling action. Notably, compared to the heuristic baselines, the latency fluctuation, spikes, and overall latency indicated by the moving average show no improvement after this pod rescheduling. After the node failures, DQN issues two pod rescheduling actions, which significantly reduce latency fluctuations and spikes, lowering the overall latency.

Upon examining the first pod rescheduling made by DQN, we found that DQN produced an invalid rescheduling action after the initial rescheduling, and therefore no subsequent rescheduling was performed. As discussed in Section 4.4.3, unlike the PPO agent used in this study, the value-based DQN agent is not able to employ the invalid action mask mechanism designed to prevent the generation of invalid rescheduling actions. Consequently, in this case, DQN was unable to produce any further optimization after the invalid rescheduling action was attempted. After the node failures, the resource availability of the cluster changed, allowing DQN to generate subsequent valid rescheduling actions and further optimize pod placement. The average end-to-end latency across the whole experiment for DQN is 241.35ms, which outperforms the other baseline algorithms.

In this scenario, PPO continues to significantly outperform the other baselines. Before the node failures, PPO issues five rescheduling actions, reducing the overall latency to around 200 ms, with fluctuations controlled within ± 20 ms. Compared to the other baselines, the latency spikes before node failures after the rescheduling process are minimal, with only one request spiking to 300 ms. After the node failures, the latency spikes to around 400 ms due to the node failures. PPO then performs three rescheduling actions to bring down the overall latency to around 225 ms. Compared to the time before node failures, the request latency fluctuation remains at a similar level, with a slight increase in request spikes, which are mostly within 300 ms. The average request latency of PPO is 220.01 ms, which is 15% less than that of the heuristic baseline algorithms and over 8% less compared to DQN.

Throughout this experiment, PPO demonstrates outstanding performance compared to the baseline DQN rescheduling algorithm and the heuristic scheduling algorithms. Not only does PPO achieve a lower average end-to-end latency, but it also effectively controls latency fluctuations that may result from suboptimal service pod placement. Furthermore, when faced with resource availability changes in the cloud-edge continuum due to node failures, the PPO rescheduling algorithm optimizes pod placement based on the remaining resources and maintains both end-to-end latency and latency stability at an exceptional level. This adaptability highlights PPO's capability to respond to dynamic changes in the cloud-edge environment, ensuring resilient and efficient performance.

Chapter 7

Conclusions and Future Directions

7.1 Overview

In this thesis, we introduced a novel rescheduling algorithm designed to optimize microservice placement within hybrid cloud-edge environments by dynamically adjusting active service placements. Unlike conventional methods that require rescheduling the entire MSA application, our algorithm performs real-time, non-disruptive adjustments, updating only partial pod placements as needed. This selective rescheduling minimizes overhead and maintains application continuity. Compared to baseline approaches, our method yields significant benefits, including reduced end-to-end latency, decreased fluctuations in request latency, and the prevention of latency spikes during node failure scenarios. These improvements underscore the effectiveness of our approach in maintaining performance stability and responsiveness in dynamic, distributed environments.

Additionally, the work presented in this thesis forms the basis of a forthcoming publication with the same title. This paper provides an in-depth explanation of our methodology for developing and implementing the reinforcement learning (RL)-based rescheduling algorithm, alongside a detailed modeling approach for Microservice Architecture (MSA) applications within the cloud-edge continuum. Furthermore, it presents a rigorous performance evaluation of the algorithm using a realistic Kubernetes cloud-edge testbed, which includes simulations of node failure scenarios to demonstrate the algorithm's resilience and adaptability. This research is being prepared for submission to IEEE Transactions on Parallel and Distributed Systems [1].

[1]https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=71

7.2 Contributions

In summary, this work makes three primary contributions. First, we introduce a novel MSA scheduling algorithm designed to dynamically reschedule microservices, maintaining optimal placement as computing resource availability changes. Unlike traditional scheduling methods that reallocate the entire MSA application in response to system changes, our approach performs minimal, targeted rescheduling steps, ensuring efficient adaptability with reduced operational overhead.

The second key contribution is the development of a reinforcement learning (RL)-based approach to derive a rescheduling policy tailored for MSA applications in a cloud-edge continuum. Previous RL-based scheduling methods have struggled to capture the complex characteristics of heterogeneous computing resources and the intricate service invocation patterns inherent in MSA applications. To address this, we model both the diverse cloud-edge computing resources and the MSA application's directed acyclic graph (DAG)-based invocation patterns, incorporating these into our RL simulation environment. This environment allows the RL agent to learn rescheduling policies that are responsive to both cloud-edge resource heterogeneity and the unique topological demands of MSA applications.

Our third contribution is the adaptation and deployment of the RL-based scheduling algorithm on a real-world Kubernetes cloud-edge continuum testbed. Given that the Kubernetes scheduler does not natively support rescheduling operations, we developed a plugin to enhance its capabilities, allowing it to monitor running applications and execute rescheduling actions as needed. We deployed the trained RL agent within this modified testbed with this plugin, and extensive experiments demonstrated that proposed scheduling algorithm consistently outperforms three heuristic baselines and one RL-based baseline. Specifically, it achieves lower end-to-end latency, reduces latency fluctuations, and eliminates latency spikes for MSA applications even under conditions of computing node failure.

Based on our works, the research questions raised in Section 1.1 can be answered:

- ***Research Question 1:*** : We model the cloud-edge continuum's heterogeneous computing resources and the MSA application's network topology. By integrating this model into the RL simulation environment and optimizing for end-to-end latency, the resulting scheduling algorithm effectively minimizes MSA application latency across the cloud-edge continuum.
- ***Research Question 2:*** RL model incorporates a rescheduling action design and a reward function that accounts for rescheduling costs, enabling it to optimize

MSA application placement with minimal rescheduling. This RL-based scheduling algorithm efficiently optimizes placement while outperforming baseline scheduling methods with fewer rescheduling steps.

7.3 Future Work

In this work, we modeled the network latency of cloud and edge computing nodes based solely on their deployment layers (cloud or edge). For future research, we plan to incorporate more sophisticated network models, such as detailed underlying network device topologies and variable network latencies between different nodes within the cloud-edge continuum. Additionally, while our current focus is on optimizing end-to-end application latency, future studies could include additional optimization objectives like bandwidth utilization, computing resource utilization, and microservice application availability. Moreover, the proposed rescheduling algorithms can be integrated with autoscaling mechanisms to jointly determine the placement of MSA applications. We intend to explore this scenario in future work, optimizing our rescheduling mechanism to effectively respond to varying user workloads in conjunction with autoscale.

Bibliography

[1] Chunye Gong, Jie Liu, Qiang Zhang, Haitao Chen, and Zhenghu Gong. The characteristics of cloud computing. In *2010 39th International Conference on Parallel Processing Workshops*, pages 275–279. IEEE, 2010.

[2] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[3] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE access*, 8:85714–85728, 2020.

[4] Sergio Moreschini, Fabiano Pecorelli, Xiaozhou Li, Sonia Naz, David Hästbacka, and Davide Taibi. Cloud continuum: The definition. *IEEE Access*, 10:131876–131886, 2022.

[5] Sergio Moreschini, Fabiano Pecorelli, Xiaozhou Li, Sonia Naz, David Hästbacka, and Davide Taibi. Cloud Continuum: The Definition. *IEEE Access*, 10:131876–131886, 2022. ISSN 2169-3536. doi: 10.1109/ACCESS.2022.3229185.

[6] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th international conference on service-oriented computing and applications (SOCA)*, pages 44–51. IEEE, 2016.

[7] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, and Minyi Guo. Adaptive Resource Efficient Microservice Deployment in Cloud-Edge Continuum. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1825–1840, August 2022. ISSN 1558-2183. doi: 10.1109/TPDS.2021.3128037.

[8] Xiang He, Hanchuan Xu, Xiaofei Xu, Yin Chen, and Zhongjie Wang. An efficient algorithm for microservice placement in cloud-edge collaborative computing environment. *IEEE Transactions on Services Computing*, 2024.

[9] Amanda Jayanetti, Saman Halgamuge, and Rajkumar Buyya. Deep reinforcement learning for energy and time optimized scheduling of precedence-constrained tasks in edge–cloud computing environments. *Future Generation Computer Systems*, 137: 14–30, December 2022. ISSN 0167739X. doi: 10.1016/j.future.2022.06.012.

[10] Samodha Pallewatta, Vassilis Kostakos, and Rajkumar Buyya. Placement of Microservices-based IoT Applications in Fog Computing: A Taxonomy and Future Directions. *ACM Computing Surveys*, 55(14s):321:1–321:43, July 2023. ISSN 0360-0300. doi: 10.1145/3592598.

[11] Lixing Chen, Yang Bai, Pan Zhou, Youqi Li, Zhe Qu, and Jie Xu. On Adaptive Edge Microservice Placement: A Reinforcement Learning Approach Endowed with Graph Comprehension. *IEEE Transactions on Mobile Computing*, pages 1–15, 2024. ISSN 1558-0660. doi: 10.1109/TMC.2024.3396510.

[12] Jayachander Surbiryala and Chunming Rong. Cloud Computing: History and Overview. In *2019 IEEE Cloud Summit*, pages 1–7. doi: 10.1109/CloudSummit47114.2019.00007.

[13] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6): 52–59, 1997.

[14] Miguel L Bote-Lorenzo, Yannis A Dimitriadis, and Eduardo Gómez-Sánchez. Grid characteristics and uses: a grid definition. In *European Across Grids Conference*, pages 291–298. Springer, 2003.

[15] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154. IEEE, 2018.

[16] Kapil Bakshi. Microservices-based software architecture and approaches. In *2017 IEEE aerospace conference*, pages 1–8. IEEE, 2017.

[17] Tomas Cerny, Michael J Donahoo, and Michal Trnka. Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review*, 17(4):29–45, 2018.

[18] Partha Pratim Ray, Dinesh Dash, and Debashis De. Edge computing for internet of things: A survey, e-healthcare case study and future direction. *Journal of Network and Computer Applications*, 140:1–22, 2019.

[19] Nathan Cruz Coulson, Stelios Sotiriadis, and Nik Bessis. Adaptive microservice scaling for elastic applications. *IEEE Internet of Things Journal*, 7(5):4195–4202, 2020.

[20] C. Centofanti, W. Tiberti, A. Marotta, F. Graziosi, and D. Cassioli. Latency-Aware Kubernetes Scheduling for Microservices Orchestration at the Edge. In *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, pages 426–431, June 2023. doi: 10.1109/NetSoft57336.2023.10175431.

[21] Ying Xie, Yuanwei Zhu, Yeguo Wang, Yongliang Cheng, Rongbin Xu, Abubakar Sadiq Sani, Dong Yuan, and Yun Yang. A novel directional and non-local-convergent particle swarm optimization based workflow scheduling in cloud–edge environment. *Future Generation Computer Systems*, 97:361–378, August 2019. ISSN 0167-739X. doi: 10.1016/j.future.2019.03.005.

[22] Panagiotis Gkonis, Anastasios Giannopoulos, Panagiotis Trakadas, Xavi Masip-Bruin, and Francesco D'Andria. A Survey on IoT-Edge-Cloud Continuum Systems: Status, Challenges, Use Cases, and Open Issues. *Future Internet*, 15(12):383, December 2023. ISSN 1999-5903. doi: 10.3390/fi15120383.

[23] Ion-Dorinel Filip, Florin Pop, Cristina Serbanescu, and Chang Choi. Microservices Scheduling Model Over Heterogeneous Cloud-Edge Environments As Support for IoT Applications. *IEEE Internet of Things Journal*, 5(4):2672–2681, August 2018. ISSN 2327-4662. doi: 10.1109/JIOT.2018.2792940.

[24] Optimal Deployment of Fog Nodes, Microservices and SDN Controllers in Time-Sensitive IoT Scenarios | IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/9685995.

[25] John Paul Martin, A. Kandasamy, and K. Chandrasekaran. CREW: Cost and Reliability aware Eagle-Whale optimiser for service placement in Fog. *Software: Practice and Experience*, 50(12):2337–2360, 2020. ISSN 1097-024X. doi: 10.1002/spe.2896.

[26] Samodha Pallewatta, Vassilis Kostakos, and Rajkumar Buyya. QoS-aware placement of microservices-based IoT applications in Fog computing environments. *Future Generation Computer Systems*, 131:121–136, June 2022. ISSN 0167-739X. doi: 10.1016/j.future.2022.01.012.

[27] Carlos Guerrero, Isaac Lera, and Carlos Juiz. Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures. *Future Generation Computer Systems*, 97:131–144, August 2019. ISSN 0167-739X. doi: 10.1016/j.future.2019.02.056.

[28] A lightweight decentralized service placement policy for performance optimization in fog computing | Journal of Ambient Intelligence and Humanized Computing. https://link.springer.com/article/10.1007/s12652-018-0914-0.

[29] A placement architecture for a container as a service (CaaS) in a cloud environment | Journal of Cloud Computing. https://link.springer.com/article/10.1186/s13677-019-0131-1.

[30] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Computing Surveys*, 54(11s):239:1–239:32, November 2022. ISSN 0360-0300. doi: 10.1145/3510611.

[31] Samodha Pallewatta, Vassilis Kostakos, and Rajkumar Buyya. Microservices-based IoT Application Placement within Heterogeneous and Resource Constrained Fog Computing Environments. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, UCC'19, pages 71–81, New York, NY, USA, December 2019. Association for Computing Machinery. ISBN 978-1-4503-6894-0. doi: 10.1145/3344341.3368800.

[32] Abbas Najafizadeh, Afshin Salajegheh, Amir Masoud Rahmani, and Amir Sahafi. Privacy-preserving for the internet of things in multi-objective task scheduling in cloud-fog computing using goal programming approach. *Peer-to-Peer Networking and Applications*, 14(6):3865–3890, November 2021. ISSN 1936-6450. doi: 10.1007/s12083-021-01222-2.

[33] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. A Deep Reinforcement Learning based Algorithm for Time and Cost Optimized Scaling of Serverless Applications, August 2023.

[34] Wenkai Lv, Pengfei Yang, Tianyang Zheng, Chengmin Lin, Zhenyi Wang, Minwen Deng, and Quan Wang. Graph-Reinforcement-Learning-Based Dependency-Aware Microservice Deployment in Edge Computing. *IEEE Internet of Things Journal*, 11(1):1604–1615, January 2024. ISSN 2327-4662. doi: 10.1109/JIOT.2023.3289228.

[35] Francescomaria Faticanti, Marco Savi, Francesco De Pellegrini, Petar Kochovski, Vlado Stankovski, and Domenico Siracusa. Deployment of Application Microservices in Multi-Domain Federated Fog Environments. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–6, August 2020. doi: 10.1109/COINS49042.2020.9191379.

[36] Shuiguang Deng, Zhengzhe Xiang, Javid Taheri, Mohammad Ali Khoshkholghi, Jianwei Yin, Albert Y. Zomaya, and Schahram Dustdar. Optimal Application Deployment in Resource Constrained Distributed Edges. *IEEE Transactions on Mobile Computing*, 20(5):1907–1923, May 2021. ISSN 1558-0660. doi: 10.1109/TMC.2020.2970698.

[37] Yihong Li, Xiaoxi Zhang, Tianyu Zeng, Jingpu Duan, Chuan Wu, Di Wu, and Xu Chen. Task Placement and Resource Allocation for Edge Machine Learning: A GNN-based Multi-Agent Reinforcement Learning Paradigm. https://arxiv.org/abs/2302.00571v2, February 2023.

[38] Wenkai Lv, Quan Wang, Pengfei Yang, Yunqing Ding, Bijie Yi, Zhenyi Wang, and Chengmin Lin. Microservice Deployment in Edge Computing Based on Deep Q Learning. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):2968–2978, November 2022. ISSN 1558-2183. doi: 10.1109/TPDS.2022.3150311.

[39] Valentino Armani, Francescomaria Faticanti, Silvio Cretti, Seungwoo Kum, and Domenico Siracusa. A Cost-Effective Workload Allocation Strategy for Cloud-Native Edge Services, October 2021.

[40] Feiyan Guo, Bing Tang, and Mingdong Tang. Joint optimization of delay and cost for microservice composition in mobile edge computing. *World Wide Web*, 25(5): 2019–2047, September 2022. ISSN 1573-1413. doi: 10.1007/s11280-022-01017-2.

[41] Muhammad Usman, Simone Ferlin, Anna Brunstrom, and Javid Taheri. A Survey on Observability of Distributed Edge & Container-Based Microservices. *IEEE Access*, 10:86904–86919, 2022. ISSN 2169-3536. doi: 10.1109/ACCESS.2022.3193102.

[42] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, November 1995. doi: 10.1109/ICNN.1995.488968.

[43] Abdullah Alelyani, Amitava Datta, and Ghulam Mubashar Hassan. Optimizing Cloud Performance: A Microservice Scheduling Strategy for Enhanced Fault-Tolerance, Reduced Network Traffic, and Lower Latency. *IEEE Access*, 12:35135–35153, 2024. ISSN 2169-3536. doi: 10.1109/ACCESS.2024.3373316.

[44] Adyson Magalhães Maia and Yacine Ghamri-Doudane. A Deep Reinforcement Learning Approach for the Placement of Scalable Microservices in the Edge-to-Cloud Continuum. In *GLOBECOM 2023 - 2023 IEEE Global Communications Conference*, pages 479–485, Kuala Lumpur, Malaysia, December 2023. IEEE. ISBN 9798350310900. doi: 10.1109/GLOBECOM54140.2023.10437143.

[45] Lulu Chen, Yangchuan Xu, Zhihui Lu, Jie Wu, Keke Gai, Patrick C. K. Hung, and Meikang Qiu. IoT Microservice Deployment in Edge-Cloud Hybrid Environment Using Reinforcement Learning. *IEEE Internet of Things Journal*, 8(16):12610–12622, August 2021. ISSN 2327-4662. doi: 10.1109/JIOT.2020.3014970.

[46] Deep Reinforcement Learning for Multiobjective Optimization | IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/9040280.

[47] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. Deep reinforcement learning for application scheduling in resource-constrained, multi-tenant serverless computing environments. *Future Generation Computer Systems*, 143: 277–292, June 2023. ISSN 0167-739X. doi: 10.1016/j.future.2023.02.006.

[48] Lin Gu, Deze Zeng, Jie Hu, Bo Li, and Hai Jin. Layer aware microservice placement and request scheduling at the edge. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2021.

[49] Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

[50] Shengyi Huang and Santiago Ontañón. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. *The International FLAIRS Conference Proceedings*, 35, May 2022. ISSN 2334-0762. doi: 10.32473/flairs.v35i.130584.

[51] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. A theoretical analysis of deep q-learning. In *Learning for dynamics and control*, pages 486–489. PMLR, 2020.

[52] Shangtong Zhang and Richard S Sutton. A deeper look at experience replay. *arXiv preprint arXiv:1712.01275*, 2017.

[53] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[54] Andrea Detti, Ludovico Funari, and Luca Petrucci. μBench: An Open-Source Factory of Benchmark Microservice Applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):968–980, March 2023. ISSN 1558-2183. doi: 10.1109/TPDS.2023.3236447.

[55] Christopher G Atkeson and Juan Carlos Santamaria. A comparison of direct and model-based reinforcement learning. In *Proceedings of international conference on robotics and automation*, volume 4, pages 3557–3564. IEEE, 1997.

[56] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N Calheiros. Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In *2009 international conference on high performance computing & simulation*, pages 1–11. IEEE, 2009.

[57] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. Edgecloudsim: An environment for performance evaluation of edge computing systems. *Transactions on Emerging Telecommunications Technologies*, 29(11):e3493, 2018.

[58] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[59] Marko A Rodriguez and Peter Neubauer. The graph traversal pattern. In *Graph data management: Techniques and applications*, pages 29–46. IGI global, 2012.

[60] Christo Boshoff. An experimental study of service recovery options. *International Journal of service industry management*, 8(2):110–130, 1997.