TU WIEN Informatics

Resource Conflict Handling in Computing Continuum using Deep Reinforcement Learning

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Vlad Popescu-Vifor
Matrikelnummer 11709668

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.-Prof. Schahram Dustdar
Mitwirkung: Dr. Ilir Murturi

Wien, 1. Juni 2025

Vlad Popescu-Vifor	Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Resource Conflict Handling in Computing Continuum using Deep Reinforcement Learning

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Vlad Popescu-Vifor
Registration Number 11709668

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Schahram Dustdar
Assistance: Dr. Ilir Murturi

Vienna, June 1, 2025

Vlad Popescu-Vifor	Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Vlad Popescu-Vifor

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 1. Juni 2025

Vlad Popescu-Vifor

Danksagung

Ich möchte meinen aufrichtigen Dank meinen Betreuern, Herrn Prof. Schahram Dustdar und Herrn Dr. Ilir Murturi, aussprechen, die mir die Möglichkeit gegeben haben, ein so faszinierendes und herausforderndes Forschungsthema zu bearbeiten. Durch ihre wertvolle Unterstützung konnte ich ein tieferes Verständnis für die Komplexität des Ressourcenmanagements in Cloud- und Edge-basierten Systemen gewinnen. Besonders dankbar bin ich für ihre kontinuierliche Begleitung und ihr konstruktives Feedback, das dieser Arbeit eine klare Richtung verliehen hat.

Ebenso danke ich der Distributed Systems Group der Technischen Universität Wien für die spannenden Lehrveranstaltungen, die mein Interesse an Cloud-Architekturen und -Computing geweckt haben. Ich bin sehr dankbar für die Bereitstellung der notwendigen Hardware-Ressourcen, die es mir ermöglicht haben, das entwickelte Framework in einem lokalen Cluster zu testen. Darüber hinaus danke ich meinem Arbeitgeber für die großzügige Bereitstellung der notwendigen Ressourcen zum Aufbau und Betrieb des Testclusters, der für diese Forschung von zentraler Bedeutung war.

Acknowledgements

I extend my deepest gratitude to my supervisors, Prof. Schahram Dustdar and Dr. Ilir Murturi, for the opportunity to explore such a fascinating and challenging research topic. Their guidance allowed me to better understand the complexity of resource management in cloud and edge-based systems. I am especially thankful for their continuous support and constructive feedback, which helped shape this thesis and provided it with a clear direction. Additionally, I thank the Distributed Systems Group at the Vienna University of Technology for offering insightful courses that sparked my interest in cloud architecture and computing. I am grateful for their provision of the required hardware resources to test the developed framework in a local cluster. Furthermore, I would like to thank my employer for providing the extensive resources needed to set up and use a test cluster, which was essential for this research.

Kurzfassung

Diese Arbeit untersucht eingehend Mechanismen zur Konfliktbewältigung in der ressourcenorientierten Orchestrierung innerhalb von Computing-Continuum-Umgebungen und schlägt eine Lösung für den Umgang mit Konflikten zwischen Ressourcenverwaltungsagenten vor. Diese Agenten sind dafür verantwortlich, Ressourcenanfragen zu bearbeiten und Ressourcen verschiedenen Deployments zuzuweisen, um eine optimale Leistung zu erzielen. Bei widersprüchlichen Anfragen geraten die Agenten jedoch in einen Konfliktzustand, was zu einer Endlosschleife von Änderungen in der Ressourcenverteilung und zu einer Verschlechterung der Leistung der betroffenen Dienste führen kann. Ziel dieser Arbeit ist es, solche Konflikte mithilfe eines Deep-Reinforcement-Learning (DRL) Modells zu entschärfen. Durch die Integration von neuronalen Netzwerken und adaptiven Reinforcement-Learning-Techniken soll die Effizienz und Stabilität des Ressourcenmanagements in der Cloud bei Ressourcenkonflikten verbessert werden.

Der Hauptbeitrag dieser Arbeit ist ein Framework, das als Blaupause für die Erstellung einer Simulationsumgebung dient, in der Ressourcenkonflikte unter Einhaltung der Kriterien des Computing Continuums gezielt ausgelöst werden können. Neben dem vorgeschlagenen Ressourcenmanagementsystem enthält dieses Framework auch ein DRL-System, das mit dem Ressourcenmanagementsystem kommuniziert, um die am besten geeigneten Optimierungsschritte für Dienste vorzuschlagen, die sich andernfalls in einem Konfliktzustand befinden würden. Der Reinforcement-Learning-Aspekt dieses Systems basiert auf der Kommunikation zwischen den Komponenten des vorgeschlagenen Frameworks. Da das System leistungs- und latenzbewusst ist, kann eine Watcher-Komponente dem DRL-Modell Rückmeldung geben, falls sich die Leistung der Dienste nach Anwendung der Entscheidung des neuronalen Netzwerks verschlechtert. Mit dieser Funktionalität kann das DRL-Modell sich auf Basis früherer Erfahrungen verbessern.

Mehrere Komponententests und Benchmarks des implementierten Frameworks zeigen, dass es erfolgreich Konflikte zwischen widersprüchlichen Agenten erkennen und Ressourcenquoten auf gesamte Nodes anwenden kann – meist ohne dabei Leistungseinbußen zu verursachen. Im Fall von Performance-Degradierung korrigiert sich das DRL-System entsprechend, um Dienste mit unterschiedlichen Schwellenwerten und Ressourcenanforderungen differenziert zu behandeln.

Die Ergebnisse der Arbeit zeigen, dass es möglich ist, ein Framework zu entwickeln, das Konflikte zwischen Ressourcenverwaltungsagenten in einer cloudbasierten Umgebung

erzeugt und diese mithilfe eines DRL-Systems löst. Dies impliziert, dass sich Ressourcen auf einem Node in einer leistungsbewussten und selbstkorrigierenden Ressourcenverwaltungsumgebung um eine festgelegte Quote reduzieren lassen.

Abstract

This research takes an in-depth look at conflict-handling mechanisms in resource-oriented orchestration within computing continuum environments and proposes a solution to conflict-handling between resource management agents. These agents are tasked with handling resource requests and allocating resources to various deployments to achieve optimal performance. However, in case of contradicting requests, the agents enter a state of conflict, leading to an endless loop of changes in resource allocations and to a worse performance of the corresponding services. This thesis aims to mitigate such conflicts with the help of a deep reinforcement learning (DRL) model. By integrating neural network models and adaptive reinforcement learning techniques, the proposed solution seeks to enhance the efficiency and stability of resource management in the cloud in the event of conflicts in resource allocation. The main contribution of this research is a framework that serves as a blueprint to create a simulation environment that allows triggering resource conflicts on services running in the computing continuum's criteria. Included in this framework, aside from the suggested resource management system, is also a DRL system that communicates with the resource management system to suggest the best-fitting optimization steps for services that may otherwise enter a conflicting state or remain running in a state that would waste more resources than required. The reinforcement learning aspect of this system relies on the communication between the components in the proposed framework. Because this system is performance and latency-aware, a watcher component is able to send feedback to the DRL model, in case the performance of the services degrades upon applying the neural network's decision. With this functionality, the DRL model can improve itself based on previous experiences.

Multiple component tests and benchmarks done on the implemented framework show that it can successfully detect conflicts between contradicting agents and apply resource quotas to whole nodes, mostly without triggering performance issues. In case performance degradation occurs, the DRL system corrects itself accordingly to treat services with different thresholds and resource requirements accordingly.

The results of the research show that it is possible to create a framework to generate conflicts between resource management agents in a cloud-based system and have them solved by the DRL system. This implies being able to reduce the allocation of the

resources of a node by a specified quota in a performance-aware and self-repairing resource management environment.

Contents

CHAPTER 1

Introduction

1.1 Background and Motivation

With the technology constantly evolving in many directions and its performance and power continuously increasing, the resource requirements and the need for improved infrastructure also increase with it. Many companies have either moved their deployments to outsourced cloud infrastructures or host the servers themselves with the help of various technologies, such as Kubernetes. Even though the last ten years were dominated by Cloud infrastructures, newer types of infrastructures, such as the Edge and the Fog, have emerged, thus increasing the alternatives of building a cloud-based system and improving the flexibility regarding resource management possibilities in distributed systems [DM21, DM20, AMD20]. A system that consists of all three aforementioned computing tiers is called a computing continuum system[DPD22]. This system tier benefits from the synergy of the three main computation tiers to offer higher flexibility and distribute resources more efficiently to workflows that are dynamically distributed based on their computation intensity.

In a computing continuum environment, resource consumption optimization is of high importance, especially when it is desired to avoid wasting resources when service workloads require less [MD22]. Resource management agents are used to alter the usage of the resources in a granularity that is otherwise not achievable by using the built-in features of existing frameworks. However, there is a critical challenge that appears in resource-oriented orchestration: the lack of effective conflict resolution mechanisms in resource allocation [TMD19]. This can lead to resource request conflicts among the management agents and system instability and increased cost of electricity. An endless conflict loop may not only waste computational resources but also reduce the performance and reliability of the deployments running on the cloud infrastructure.

Kubernetes is indeed a very powerful orchestration tool, however it has no innate functions that cover optimizing resource usage for the deployed services. Additionally,

in a computing continuum environment, a user would not want to pay for the cost of resources that are allocated but not used. This is where the agents come into play, and thus also the conflicts between them. Solving these conflicts is of high importance for maintaining the resource efficiency of the cluster and optimizing the performance of its services, as well as maintaining user satisfaction. Therefore, this research aims to develop and integrate advanced conflict-handling strategies for mitigating such scenarios, which may occur between agents.

1.2 Use Case Scenario and High-Level Problem Description

The main use case for this research will focus on RAM optimization for reducing the overall cost consumption of a node within the cluster environment while prioritizing the performance and response time of the services. Because resources are by default only allocated and not optimally set based on the service requirements, agents are created to handle this problem. Each agent is responsible for optimizing a specific resource, be it the number of CPU cores or the energy cost. The agents, after obtaining a resource optimization request from the administrator, communicate with a coordinator microservice named "watcher" that applies the suggested changes. In this case, two resource suggestions made by two different agents may be contradictory, leading to a conflict. As an example, a request to decrease the cost of the power consumption of the services may conflict with a request that demands an increase in memory among some services. This conflict leaves the agents in a loop, each fighting over their resource suggestions since the desired goal cannot be achieved.

In a Kubernetes cluster, there are several services deployed on the same node. Requests can be sent to agents for scheduling specific allocations regarding the desired resources:

1. **Resource Optimization Request**:
   - The administrator wants to decrease the average cost per hour of the services running on node A. Therefore, they send an optimization rule to the power usage agent to minimize the amount of resources used by the services in a specific node while respecting the resource thresholds of the services. We assume that the CPU/RAM thresholds of the services are already known.
   - The coordinator, after receiving the request from the costs agent, reduces the CPU and memory resources among the services. The optimization rule would frequently re-check the cluster situation to re-optimize it.
2. **Second Request from Service User:** A user would like additional CPU cores for service A, which also runs on node A because for the next day, there will be more complex computation requirements. Therefore, a further resource request is sent to the agent to provision extra cores to service A.

3. **Endless Conflict Loop**:

   The agents, operating without a global conflict resolution mechanism, continuously attempt to execute the overlapping requests. While the admin's request will keep the resources low among all running services on node A, the user's request will continuously try to allocate the additional resources to service A, but not successfully. As a result, a conflict loop is triggered between the agents and the desired performance of service A is degraded due to the continuous switching of CPU cores from the conflicting requests of the two agents.

This leads us to two types of conflicts that occur in this system. The first one is a conflict within the agent specifications on the configuration level, and the second one is a conflict of optimization on the infrastructure level, which results in the worsened performance of the services. To mediate these conflicts, a versatile DRL model using various techniques, such as prioritizing being aware of the services' requirement thresholds and performance statistics would be of high importance.

1.3 Research Questions

The main contribution of this research will be creating a deep reinforcement learning (DRL) model that can resolve resource allocation conflicts between agents in the computing continuum. Further automatic training of the deep learning model using a reward system is an interesting approach for solving conflicts in resource management. Therefore, the research question will be structured as follows:

1. *How can deep learning models, using synthetic conflict generation and benchmarking, be trained to manage resource conflicts while being aware of problem-solving latency in the cloud environment?*

2. *How effectively can a deep learning model mediate resource management conflicts caused by agents in the computing continuum, and how can its performance be measured and iteratively improved using outcome-based feedback?*

3. *What metrics can be used to benchmark the efficiency and impact of their resource allocation decisions?*

1.4 Structure of the Thesis

This research will consist of a thorough literature review of the current resource management situation in the computing continuum, as well as an analysis of the existing resource management models and algorithms. Afterwards, a framework will be suggested that will encompass the complete resource management system, as well as the DRL system. After clearly defining the interaction between the components and their functionality

in the framework section, a cluster architecture similar to the cloud will be set up for the research environment, where the components discussed in the framework will be deployed and set into motion. Afterwards, the performance of the proposed system will be analyzed through benchmarks and evaluations, followed by the conclusion.

This thesis will contribute a framework that allows solving resource allocation conflicts:

1. First, a cluster will be created for the research, on which services will be deployed across the nodes.
2. Afterwards, the agents will be designed, which will receive the resource allocation requests from the administrator and forward them to the watcher. Each agent will be responsible for a specific resource, such as the CPU cores or the RAM.
3. The watcher will be created so that it will have the ability to orchestrate the agents and apply their resource specifications to the cluster. This will be the starting environment for this research, where conflicts can be triggered between the agents.
4. The DRL system will then be deployed to handle the resource conflicts and perform optimization tasks.

The DRL System will be used to compute the solution to the resource management conflict problem by using a neural network model and a reinforcement learning approach. In combination with a preset prioritizing of resources, the model will train itself consistently based on the feedback obtained from its past suggested solutions. This means that if one decision is wrong and the same conflict appears again, the model would take a different approach and suggest a better solution.

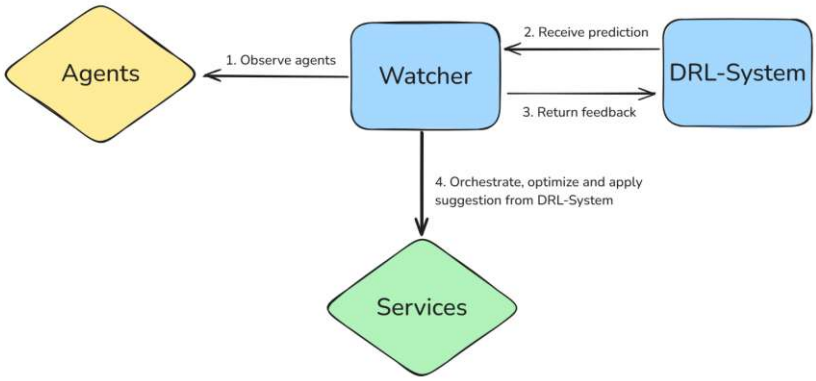

Figure 1.1: Overview of the communication between the components.

The communication between the components would look as depicted in Figure 1.1:

1. The watcher oversees the activated agents and coordinates and applies their resource specifications to the environment.

2. The watcher constantly communicates with the DRL system and receives optimization or conflict solution steps for the relevant services.

3. In case of bad predictions, the watcher sends feedback to the DRL system to improve itself.

4. After the resources have been allocated as suggested by the agents, the watcher keeps track of the optimized services' performances and returns a rating to the DRL system, depending on how the assessed allocation performed.

1.4.1 Concept of the Deep Learning Functionality

To enable a deep learning reactive system, it must consist of a state, a communication to the watcher, and the neural network model itself.

- **The state** will allow the DRL system to keep track of the cluster's resources and their allocation, as well as a history of the previous requests made by the agents.
- **The communication with the watcher** will be required to send the suggested conflict solution and apply it to the environment.
- **The neural network model** will be used for complex conflicts that occur in optimization. This deep learning model will be thoroughly trained to solve possible conflicts and receive feedback based on the decisions taken. This enables the self-improvement of the model using reinforcement learning.

1.5 Scopes and Limitations

The aforementioned approach will be expanded upon to ensure adaptability to any possible outcome of the feedback, meaning that the DRL system would have to react again and send a better resource allocation if it received a bad rating on the previous decision. For this, a logical structure has to be designed, as well as the ability to dynamically prioritize resources based on various factors. The overall evaluation of the model will be done using benchmarks and test suites. In case the model can't find a viable solution for the conflict, the admin will be informed of the incident. A viable solution implies a solution that acts as a mediator between the conflicting agents and does not affect the performance of the services to which it is applied.

Relevant resources for this research are the number of CPU cores, RAM, and storage space. Other metrics will be derived, such as cost estimations or average energy usage per hour.

The expected results consist of building the aforementioned architecture with its components and building the DRL system to solve the conflict problem in resource allocation. Additionally, achieving a significant success rate in versatile conflict-solving using the suggested approach is of high importance. A test suite will be used to evaluate the DRL

system, and various benchmarks will be used to evaluate the neural network models themselves, for finding the best-performing model to use for solving conflicts.

The following limitations may occur in this research, which would in turn define its scope:

- The proposed architecture will be developed in a virtual Kubernetes cluster. In this case, the limitations are those already included in the Kubernetes framework and those that lie within the hardware on which the virtual machines will be deployed for the cluster nodes.
- The complexity of the agents, watcher, and consumer services will be minimal since they are not the main focus of the research.
- Because the DRL system and watcher would, in theory, require many permissions for the cluster, the security aspect of the implementations will not be actively researched.
- Lastly, security and privacy aspects are not considered within this work. However, different approaches as presented in [BMDD23, MDP+23, KMD24] can be implemented easily.

CHAPTER 2

Literature Review

2.1 State of the Art

In the past few years, the research in the cloud domain has increased significantly, with many scientific papers tackling the resource management aspect of the cloud infrastructure. As mentioned in the works of S. Dustdar, V. Pujol et al., the cloud continuum aims to create a dynamic and adaptive computing environment that spans between cloud and edge computing [DPD22, CPDM+23]. The resource perspective in such a system is essential because the computing continuum depends on the resource availability for the applications that run on it [MATD19, MD21, MD22]. An important issue mentioned in this research is the reactive management system, which, while fitting for other systems, would not be suitable for computing continuum systems. A reason for this is the dependence on the various configured thresholds that are defined in the system [DPD22].

Some contributions include deep learning and reinforcement approaches. However, they focus on resource allocation and management, not reactive interventions if agent conflicts occur. An example of such an approach would be the suggested deep reinforcement learning model by D. Zeng et al., which proposes a model-free solution to automatically predict user resource consumption patterns based on historical interactions [ZGP+19]. While this contribution benefits from a model-free solution, it is constrained to the environment it is deployed on, because the deep learning system is trained with the infrastructure's data. However, the interation between the environment and the neural network model for reinforcement learning is relevant to this thesis. Additionally, the research held by N. Naderializadeh et al. suggests a deep reinforcement learning distributed model, which runs across multiple agents. This paper describes a more complex deep learning model that depends on the communication between the distributed agents [NSSN21]. An important aspect of this paper is that the agents are independent and may make decisions simultaneously while being unaware of the other agent's decisions. This

is within the scope of this thesis because if agents are unaware of each other's choices, conflicts will appear between them.

An interesting DRL implementation is documented in research by Liu et al., which uses Q-Learning to derive the optimal choices when facing specific situations. Q-Learning is an algorithm used in reinforcement learning that makes use of tables that contain a specific rating for various combinations of states that may appear in the environment [LLX+17]. With the help of this feature, it was possible to improve a neural-network-based framework that is used in solving resource allocation and power management problems in cloud computing.

The research by Guo et al. describes a downside of neural network training, which is the over-generalization and negative knowledge transfer that may occur when a neural network is used for multiple tasks. To solve this issue, a branching of the network in a tree-structured design is proposed, which segments the whole deep learning system into components that are focused on specific tasks [GLU20]. This insight is relevant to this research since the number of available deployments with different resource requirements and performance thresholds may increase the complexity that needs to be trained to the DRL system.

Yi Wei and M. Brian Blake tackled a resource management problem in the cloud in their research. Before proposing a framework, they mention the main challenges that arise in a cloud's resource management. The primary question is how services can be ranked without knowing their implementation details. Afterwards, they mention the problem of automatically determining the resources allocated to a service based on its requirements. An important challenge mentions managing the allocated resources of a service at runtime, so that the owner would not have to do it manually[WB13]. While these statements are based on different infrastructure and technology, they are still relevant to this research because such control over the granularity of resource management in the edge infrastructure is also in a state where it requires improvement. Although it is a bit older, this research offers a perspective on resource management before deep learning reached its maturity and started to be used in many management systems.

Furthermore, research has been conducted to predict cloud environments' workload and resource consumption using deep learning models [AABAH22, BLYZ21].

Table 2.1 summarizes the comparison of the related work with information on the focus of the research, the methodology used, the limitations, and the relation of the paper's contribution to this thesis.

2.2 Literature Review for the Framework

To prepare the best corresponding framework for the resource management problem, several relevant articles have been analyzed to view the best practices regarding the underlying architecture, the networking between the microservices, and the existing implementations of resource agents in the cloud.

Work	Focus	Technique	Limitation	Relation to This Work
On Distributed Computing Continuum Systems[DPD22]	Overview of computing continuum systems	Conceptual modeling of distributed architectures	N/A	Motivates need for conflict-aware resource orchestration in continuum environments
Resource Management at the Network Edge: A Deep Reinforcement Learning Approach[ZGP+19]	User resource demand prediction at the edge	DRL for user consumption forecasting	Model-free but constrained to environment it is trained in	RL-based improvement through environment-to-neural-network communication
Resource Management in Wireless Networks via Multi-Agent Deep Reinforcement Learning[NSSN21]	Decentralized wireless resource allocation	Multi-agent DRL (independent learners)	No explicit conflict resolution; agents are not aware of each other	Uses DRL but lacks inter-agent conflict consideration
A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning[LLX+17]	Resource allocation for energy efficiency for best power-latency tradeoff	Tabular Q-learning in a hierarchical model	Different definition and tasks for agent	Split logic between VM and local server layers
Learning to Branch for Multi-task Learning[GLU20]	Multi-task DRL performance	Tree-structured branching model	Not designed for resource orchestration or distributed environments	Inspired branching concept reused for feedback adaptation in this work
Adaptive Service Workflow Configuration and Agent-based Virtual Resource Management in the Cloud[WB13]	Service workflow management	Agent-based workflow management framework using proposed algorithm	Older infrastructure and technology	Relevant example for resource management on runtime achieved using algorithms
This Research	**DRL solution for conflicts between resource management agents**	**Adaptive DRL with feedback-based branching**	**Synthetic data training;**	**Extends existing work by offering possibility to simulate resource management conflicts between agents and solve them using a DRL approach**

Table 2.1: Comparison of the related work.

2.2.1 Environment Architecture

There are many tools available for simulating cloud environments, many of which have been kept up-to-date in recent years. These tools allow evaluating services in a flexible and controllable environment before deploying them in a real cloud environment where resource costs would apply[CR+11].

Many contributions from various research papers were based on CloudSim. CloudSim is a Java-based framework that supports modelling and simulating virtualized cloud infrastructures based on virtual machines. It has been created with the main goal of

simplifying the testing of the resource usage of cloud services in a closed environment that allows testing and reproducing workflows for calibrating and optimizing the performance of the services[ASY20, CK13]. An Edge variant has also been developed, which builds upon Cloudsim to simulate edge computing scenarios [SOE18]. The research paper *"Towards improving resource management in cloud systems using a multi-agent framework"* by Mahmoud Al-Ayyoub et. al[AADJA16] uses a CloudSim-based architecture for building a framework to optimize resource provisioning and utilization by monitoring the service user's behaviour and resource usage patterns in virtual machine(VM) provisioning. The multi-agent framework consists of local utility agents, which estimate the amount of resource usage of a user based on the usage history and create a resource usage profile that would then be used to estimate a more fitting VM specification and a global agent which receives the specifications from the local agents and is responsible for the provisioning of the resources[AADJA16].

While tools like CloudSim proved to be very useful in research and in testing the performance of cloud-based services before the productive deployment, Kubernetes also excels in setting up stable, cloud-based environments. While it may have a steep learning curve for the normal user, its accuracy and performance in setting up clusters consisting of multiple nodes are why it is one of today's go-to technologies for cloud services. Kubernetes has emerged as a leading open-source platform for managing containerized applications in a distributed environment. Unlike traditional cloud simulators like CloudSim, which focus on virtual machines, Kubernetes is designed to orchestrate containers, offering scalability and resilience across cloud-native applications. Through Kubernetes, users can create and manage container clusters that automate deployment, scaling, and operations consistently and efficiently. This platform provides significant control over resource allocation, load balancing, and fault tolerance, making it ideal for testing applications' performance and resilience within a controlled setting. By utilizing Kubernetes in conjunction with other simulation tools, researchers and developers can comprehensively understand their cloud applications' behaviour in diverse infrastructure configurations, enhancing their robustness before real-world deployment.[Ku24]

Minikube was created for smaller test cases. This open-source project allows deploying a local instance of Kubernetes, which may support different providers, such as Docker, Qemu, Hyper-V, or KVM. This local environment supports the latest Kubernetes releases and only requires at least 2 CPUs, 2GB RAM, and 20GB of free disk space [The25]. Another distribution worth mentioning is K3S, a lightweight Kubernetes cluster that allows running production workloads in resource-constrained environments, such as the Edge. An important highlight of this distribution is that it is also optimized for ARM architectures, allowing it to run on devices as small as Raspberry Pis[K3s].

2.2.2 Autoscaling and Provisioning

A significant detail is the importance of dynamic resource provisioning in cloud infrastructures while considering the user's quality of service (QoS) requirements for the workload. Using this technique, it is possible to adapt the allocated resources dynamically to a service

based on its workload to avoid either underprovisioning or overprovisioning[AADJA16]. While underprovisioning is more tangible because it has observable effects on the performance of the service, overprovisioning of resources leads in the long term to increased costs for resources that were not required. This concept is relevant for this research because prioritizing aspects of the service, such as latency, is essential for building a well-performing conflict solver. Kubernetes supports automatic scaling of services based on their resource usage with the help of the Horizontal Pod Autoscaler. It works by creating additional replicas of the service so that it is horizontally scaled[Ku24]. While this method is effective, it may cause a big increase in available resources, where only a smaller increase of e.g. 10% would suffice. This is because Kubernetes requires predefined requests and limits of resources to be included in the deployment YAML configuration. If, for example, a pod has a limit of four cores and uses only one, all four cores will be assigned to be used only by this deployment, even if they would not be actively used in a computation. In this case, upscaling the service and iteratively decreasing the core limit on each pod would solve the overprovisioning. In this case, upscaling is important since updating the resource limits and requests requires rebooting the container. To automate this process, Kubernetes developed the Vertical Pod Autoscaler as a side project. It is not included in Kubernetes by default; however, it can be set up to scale services vertically by using its Custom Resource Definitions (CRD). For the VPA, they are used to define when and how to scale the resources of the replicas of a deployment[Ku24].

Another scaling method included in Kubernetes is Cluster Autoscaling, which supports automatic horizontal scaling of the nodes of the cluster[Ku24]. This method, however, is not within the scope of this research.

The three aforementioned scaling methods supported by the Kubernetes infrastructure describe the flexibility available for the resource management simulation environment. Aside from direct changes in the services' resource requests and limits, scaling may also be an efficient solution for dynamically altering the available resources based on the actual workload.

2.2.3 Monitoring

Monitoring and observability are two very important factors for this research, and any cloud infrastructure must fulfill all the criteria to be used in a production environment. While monitoring collects and analyzes data about the performance and resource consumption of the services running within the cluster, observability offers more insight for the administrator by granting a more human perspective of the internal behavior and state of the system.

By default, Kubernetes comes with the Metrics API. However, it is more limited because it keeps track only of the CPU and memory usage[CYY+17]. Because of its popularity in the research area and enterprise domain, Prometheus has also been analyzed for this thesis. The research *"Observability in Kubernetes Cluster: Automatic Anomalies Detection using Prometheus"* by Octavian Mart et. al.[MNPC20] contributes with an approach to

improving the already existing monitoring system in the Kubernetes cluster by training a prediction model to forecast Prometheus metrics and pushing them back to Prometheus. This improves the ability to detect resource anomalies better and improves the overall observability of the cluster [MNPC20]. Prometheus works via a pull strategy, which pulls data in a specific interval from targets within the cluster and processes the data into a time series[MNPC20]. A very important feature of Prometheus is that it allows the creation of alerts, which may get triggered if a specific threshold of a service's consumption is reached. Additionally, it also allows tracking the latency of the systems and offers machine-level metrics. For observability, Prometheus offers a dashboard, however, it is not widely used because of its limited functionality. On the other hand, Grafana is a popular app for building and visualizing dashboards [MNPC20]. Research has been done in designing the best combination of metric gathering and observability services, and Prometheus with Grafana is an important stack with a powerful synergy[SB19]. Because of the great performance in offering a monitoring solution to complex infrastructures, many Kubernetes distributions have adopted this stack combination by default, such as RKE2 by Suse[rke].

As an alternative to Prometheus, Heapster is also used as a cluster-monitoring tool in Kubernetes and offers similar features[CYY+17]. Both of the aforementioned technologies go hand in hand with InfluxDB for saving the output time series and having them further managed by the administrator or processed by Grafana[CYY+17].

InfluxDB has been designed to store time series data and performs better than relational databases since they are not optimized for common time series workloads. InfluxDB can store large volumes of time series data and quickly perform operations on it. An important feature of this database is that it uses a timestamp to identify a single point in a data series, and it also offers a flexible field-based schema which can be extended at any time by adding new entries[Inf24].

2.2.4 Deep Learning

Deep learning has immensely evolved in the past few years in terms of framework variety and performance because of new hardware with higher computation power. TensorFlow is one of the leading frameworks for creating and training deep learning models. It is an open-source project that includes the functions and operations required for creating models. While TensorFlow supports training models using only the CPU, it also offers GPU support[ZTB+18]. Training deep learning models using GPUs has become almost a standard because of the significantly increased training speed for complex models, which would take hours to train when using only one CPU. The research by Lawrence et al. compares TensorFlow models on different hardware and shows the significant improvements in training models using GPUs [LMR+17]. TensorFlow allows access to the Keras framework, which simplifies the creation of a neural network by providing a means of configuring the model while also enabling modularity. Its library includes definitions of layer classes used to shape the neural network, thus tailoring it to fit best for the required task [PNW20].

With the number of frameworks available for building neural networks, finding the best-suited library for a use case has become a more complex task. PyTorch is another well-known framework that also excels in deep learning model creation. A comparison has been conducted between PyTorch and TensorFlow. Results show that PyTorch offers a more user-friendly interface and requires fewer steps to set up, which makes it more suited for beginner-friendly environments. It also has faster training and execution rates. On the other hand, TensorFlow relies more on accuracy, offering more control over the whole setup, which in turn may lead to an increase in complexity[NCN+22].

CHAPTER 3

The Conceptual Framework

In this chapter, we first define the main features and goals of the framework, followed by the overall system architecture of the proposed approach. Then, we discuss each essential component of the framework in detail. Afterwards, we explain in detail the structure of the deep learning conflict-solving solution.

3.1 Main Goals and Features

Although many resource management systems have been created for managing various systems based on their requirements, the primary goal of this framework is to first provide the possibility of simulating an environment on which conflicts in resource management can be manually triggered. In other words, this framework provides not only a DRL-powered resource management system but also a conflict simulation mechanism for setting up a simulation environment.

The main features that distinguish this framework from others are:

1. The possibility to easily simulate cases where resource management agents are conflicting.
2. Having the ability to easily revert the services from any applied agent resource configuration to reset the simulation environment to the initial state. This allows re-testing various scenarios by always starting from the initial point.
3. An iteration-based algorithm for solving conflicts between agents in case their resource management goals overlap in their specifications.
4. A performance- and latency aware, DL-powered decision making system for resource usage optimization based on quotas.

5. Reinforcement learning for DL model adaptability to special thresholds in case of false optimization suggestions.

This framework also takes a different approach with regard to the structure of the resource management agents. Instead of having different agent services communicating to a coordinator, the agents are now defined as specification objects, with their functionality integrated into the coordinator (the watcher). This centralized design can offer advantages in certain scenarios, especially when it comes to better coordinating the agents or saving computation resources due to fewer required containers.

3.2 System Architecture

Figure 3.1 illustrates the main components of the proposed framework and how they communicate with each other. As mentioned in the background section, the framework will consist of the following parts:

1. The agent controller, which serves as a gateway that receives input from the admin containing the specification of the resource quota and translates it into an agent object.

2. The agent which is a specification that is saved in the state of the cloud and serves as an independent depiction of a resource management agent's task. It describes a configured resource specification that needs to be applied by the watcher.

3. The watcher oversees all agents' specifications and applies them to the targeted services or nodes, depending on the specifications, with the assistance of the deep learning model. The watcher is resource and latency-aware since it is connected to a metrics API.

4. The metrics API is the component that tracks all the running workflows within the cloud infrastructure and provides a means of storing the metrics and accessing them using an API.

5. The deep reinforcement learning system, which communicates with the watcher to offer solutions to occurring conflicts and scaling opportunities while managing the trained DRL models.

All these components form the custom resource management system, which oversees resource usage and receives resource orders from the administrators, which are then applied to the underlying cloud infrastructure.

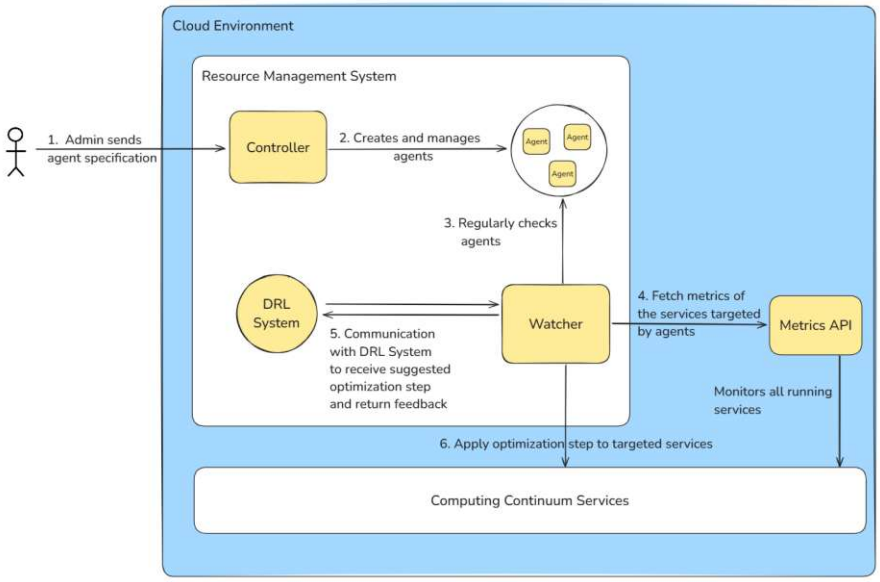

Figure 3.1: Workflow and communication of the proposed conceptual framework.

3.3 Computing Continuum Services

Before diving into the details of the resource management system, its main target has to be defined first. This framework's primary goal is to optimize computing continuum workflows, and, based on the definition, upon optimization, the workflow may not be interrupted, and the computation must continue. For this, the consumer services running in the underlying cloud infrastructure must be configured to update themselves using a *"RollingUpdate"* strategy, which does not interrupt the workflow.

The choice of the functionality of these services is free for testing use cases; however, for evaluation, it is required to have similar consumer services deployed, which complete similar tasks continuously or have them ranked in groups based on their performance or average resource usage. A critical aspect in the optimization process of the consumer service is the resource threshold, which, in turn, would have to be learned by the DRL system using reinforcement learning.

3.4 The Resource Management System

The resource management system, along with the deep learning conflict solver model, is the main contribution of this research, which tries to resolve the problem of the lack of dynamical conflict-solving systems for compute continuum workflows. It consists of more components that communicate with each other, thus acting as a main hub that is aware of the cluster's state and resource usage for the running apps. All the components and services discussed in this framework are container-based deployments within a cluster. The agent controller is the first component, which enables the administrator to set a

resource request for the cluster. The controller, after receiving as input the resource request specification from the administrator, creates an object with the corresponding information and pushes it into the cluster's state, where it can be reached by other services if needed. This object is called an agent and, in this particular example, is a custom resource definition (CRD) in the Kubernetes cluster, which is tailored to this research's needs. The next important component is the watcher, which oversees the state of the cluster and reacts when new agents are created or changed by applying them to the cluster. Simply acting on the specifications sent by the administrators often leads to conflicts which may lead to underprovisioning or overprovisioning, as well as decreases in the SLOs of the services, such as increased latency or computation time. This is where the deep learning model comes into play, which, after being correctly trained, receives information about the resource changes from the watcher and suggests steps to be taken to prevent conflicts. With this overview in mind, each component can now be described in more detail.

3.4.1 The Agent

An agent is an object of a specific structure that represents the specifications and the state of the orders published by the controller. It is highly important that it is saved in a state so that the responsible components can constantly view and update it. It is created and managed by the agent controller. Its main goal is to provide a resource specification that would then be applied to the cluster by the watcher.

The structure of the agent looks as follows:

- **The scope**: Describes on which scope the specification should be applied. It can either be set to *"deployment"*, *"namespace"* or *"node"*.
- **The status**: Describes the actual state of the agent. This may be *"pending"*, *"applied"*, *"conflicting"* or *"disabled"*.
- **Target name**: Depending on the scope, the name of the node, deployment, or namespace has to be provided.
- **Resource quota**: This may include the desired resource quota to be applied in the cluster, depending on the scope.
- **Restart count**: Keeps count of how many times the agent's specification had to be reapplied.

The agent has been designed in this way so that the administrator can offer more flexibility when setting resource management rules and storing them separately as centralized specifications simplify communication between the agent controller, the watcher, and the deep learning model. This also implies that the agent's functionality is also separate from the specification object. In this case, the agent's functionality is defined within

the watcher component. The scope parameter completely changes the agent's goal, for either applying the rule on a specific deployment, namespace, or whole node. The state parameter grants the user insight into whether the agent has been applied correctly or not. The *"conflicting"* state is then triggered if the agent has to be reapplied on each check iteration made by the watcher. In this case, a counter is kept to oversee the length of the conflict. These conflicts are logical conflicts that can happen between conflicting agents and not the more complex conflicts that have to be solved by the deep learning model.

3.4.2 The Agent Controller

The agent controller acts as a gateway for resource requests and receives HTTP inputs from the administrator. Upon validating and verifying, the controller uses the received resource specification and creates the agent object. This implementation can utilize a web framework such as FastAPI to handle HTTP requests and, upon establishing a connection to the cluster, create, edit, or delete the agent depending on the request received. The endpoint *"/set-limits"* allows the admin to define granular resource constraints at various scopes, including nodes, namespaces, and deployments. This flexibility is critical in edge environments where diverse workloads compete for finite resources. The system processes the structured requests, ensuring data validation and schema conformity. The following algorithm illustrates the high-level functionality of the agent:

Algorithm 3.1: Agent Controller

Require: Request with resource specification (scope, node, memory quota)
Ensure: Create the agent objects based on the received specification

```
1: Wait for requests to set agent specification
2: Define agent with request data
3: if agent exists then
4:     Update existing agent with new resource quotas
5: else
6:     Create a new agent with the received specification
7: end if
```

In addition to its main functionality, the controller offers a dashboard that serves as a centralized way of overseeing the actual impact of the agent specifications on the infrastructure's resources. With a combination of API connections, the agent controller can provide a platform that provides up-to-date information regarding the existing agents, the deployments they have been applied to, as well as graphs that display their evolution in resource consumption.

An LLM-powered input variant is also included as a feature to simplify the specification input on the dashboard service provided by the controller. With the help of an LLM API connection, it is possible to configure an endpoint with a predefined prompt to

extract the required specification details from normal text and place them in the required structure object.

3.4.3 The Watcher

The watcher is the more complex service in this setup when comparing it to the controller because it oversees the ongoing and pending agents and works as a coordinator and checks if the agent specifications have been applied successfully. It is connected to the cluster's API, so it is always aware of the states of the existing agents and ensures that they are also applied to objects newly created within the scope. The watcher is also aware of the resource consumption of the services targeted by the agents and can act and apply resource changes as configured in the agent specification. Because of the watcher's importance when sending information to the decision-making, deep learning model, it is also performance-aware and has an overview of the resource consumption of the deployments in the cluster. This can be achieved by connecting the watcher to the API of a metrics service that keeps track of this information.

The agent's functionality is defined within the watcher, as it applies the resource quota depending on its defined scope. For each type of scope, the watcher has a different technique for applying the agent's specification:

- For the **deployment** scope, the watcher fetches the deployment object using the name included in the agent object and applies the specified resource quotas.
- The **namespace** scope first lets the watcher obtain all existing deployments within the namespace and apply the resource specification from the agent to each of them.
- The **node** scope focuses on searching deployments that have the specific node as a nodeSelector and are not important for the system. This use case should replicate a case in edge computing where the overall consumption of the edge device has to be decreased.

The watcher works iteratively while checking the existing agents and if they have been successfully applied, every five seconds.

Regardless of the agent's scope, if its specification has been successfully applied to the target deployments, the watcher switches its state to *"applied"*. While in the *"applied"* state, the agent is regularly re-checked by the watcher, and if changes in the target deployments' resources can be found, the agent is reapplied to prevent inconsistencies. This is one important mechanic that makes the simulation environment work. At this stage, it is now possible to create the agents and have them be applied to the cluster's state with frequent checks in case they need to be reapplied. This also implies that conflicts may appear if the complexity of the agents increases. Algorithm 3.2 summarizes how the watcher tracks the agents and applies their resource quotas. This includes communicating with the DRL system and overseeing the performance of optimized deployments. It is

important to mention that an agent that does not apply any quotas only triggers the optimization aspect of the watcher's workflow. This means that in this case, the watcher would simply allocate the resources to the services based on their consumption, thus reducing the reserved resources and resource limits if they are not in use.

3.4.4 Deep Reinforcement Learning System

This component represents a service that provides access to a neural-network-based model that, after being trained in a specific way, can decide, based on particular criteria, the correct step needed when presented with a deployment that an agent targets. The DRL System is mainly based on a meta deep learning model, which is trained to understand optimization possibilities for deployments with average resource requirements and thresholds. However, if the model suggests a wrong optimization step, a clone of the meta model is created. This clone learns the correct step and will be used only for this deployment for future predictions as a specialized model[GLU20].

Meta DRL Model

The deployment object structure that is used as an input when training the model looks as follows:

These features allow the model to make decisions based on latency and resource consumption differences. The watcher already provides the memory and latency parameters since it is connected to the API of a monitoring tool, such as Prometheus. The initial values are the latest metrics of the deployment before applying the agent's specification. The applied values are the values that have been calculated by the watcher for the target deployment to respect the set resource rules of the agent. The deep learning model also has to take into account the available replicas for the deployment to be able to suggest scaling operations. It is important to mention that the train sets should be prepared in such a way that the model can learn the average resource thresholds of containers.

The optimization step is classified into the following parameters:

- 0: Optimize (No changes in orchestration),
- 1: Migrate to another node,
- 2: Scale down,
- 3: Scale up.

The optimization steps correspond to the deployments if specific criteria are fulfilled. For example, a pod of a deployment should be migrated to another node if the latency is significantly higher, OOMKill is reached, or the amount of allocated memory is so low that the container can no longer perform correctly. When deciding steps regarding scaling, the deep learning model is trained to take into account the number of replicas. The

Algorithm 3.2: Watcher

Ensure: Apply resource limits based on the available agent specifications

```
Start watching agents
for each agent do
    if agent.state = "pending" then
        Calculate required resource changes
        Send deployment object with required resource changes to the DRL System and
        await optimization suggestion
        Apply optimization step received from DRL-Model
        agent.state = "applied"
        agent.reapply_count = 0
    else
        Check if agent resource specification is still applied to targets
        if Check returns "false" then
            Reapply agent specification to target objects
            agent.reapply_count += 1
        else
            agent.reapply_count = 0
        end if
        if agent.reapply_count > threshold then
            agent.state = "conflicting"
        end if
        if agent.state = "conflicting" and agent.reapply_count = 0 then
            agent.state = "applied"
        end if
        Check performance for each target service optimized by the agent
        for each target service optimized by agent do
            if service performance degraded beyond the threshold then
                Revert the configuration of the service to the initial state
                Mark service as "high priority"
                Migrate service to another node
                Send negative feedback to the DRL System with the correction
            end if
        end for
    end if
end for
conflicting_agents ← list of all agents where agent.state = "conflicting"
if conflicting_agents not empty then
    Sort conflicting_agents by creation_timestamp descending (latest to oldest)
    Set state of the first (latest) conflicting agent to "disabled"
    for all remaining conflicting agents do
        Set agent.state = "pending"
    end for
end if
Sleep for 30 seconds
```

Field Name	Description
replicas	Integer
initial_avg_cpu	Float (MB)
applied_cpu	Float (MB)
initial_avg_memory	Float (MB)
applied_memory	Float (MB)
initial_avg_latency	Float (Seconds)
applied_latency	Float (Seconds)
optimization_step	Integer (0,1,2,3)

Table 3.1: Description of the training input features.

memory parameters include the memory consumption for the whole workflow, meaning the sum of all replicas. If, for example, two replicas of a deployment use 100 MiB each and the applied_memory implies a total consumption of 320 MiB, the model would then suggest scaling up.

This approach was chosen for this DRL model to promote its simplicity, making it more stable and interpretable when it comes to suggesting optimization steps for deployments based on their current resource usage and performance. Having these steps more abstract grants the watcher more flexibility when it comes to applying the suggestions made by the DRL model.

3.4.5 Reinforcement Learning System

Any cloud-based infrastructure has services with different resource requirements and thresholds. This is mainly because of the different service architectures and computation tasks that must be performed. In a well-established and running infrastructure, these services may already have their resource requirements configured and thresholds documented. However, they may not be visible to the resource management system. Because of this, the DRL model, which is trained on more average resource consumption thresholds, may suggest optimization steps to services that may not fully correspond to the service's actual resource threshold and lead to worse performance. To tackle this problem, the DRL model has to learn from these mistakes and remember to act correctly in the future for these deployments.

To enable such functionality, an instance-based reinforcement learning method is implemented in the DRL system. Based on the main DRL model, which serves as a meta model that works with thresholds that most deployments may accept, this reinforcement learning system clones the meta DRL model and, in case negative feedback is received from the watcher and re-training is required, saves it as a deployment-specific model. This specific model includes the adapted thresholds for optimization decisions and will

be used again for future predictions for this specific deployment. Figure 3.2 describes the decision flow in detail. With the help of a dictionary, the DRL system container keeps track of the specialized models that are further trained specifically for the corresponding deployment. In case such a specialized deployment is not available, it is created by first cloning the meta model, then training it with the suggested correction and reward score included in the feedback. Afterwards, the specialized model is saved in a persistent volume, so it can be accessed even after the container restarts to restore the training data.

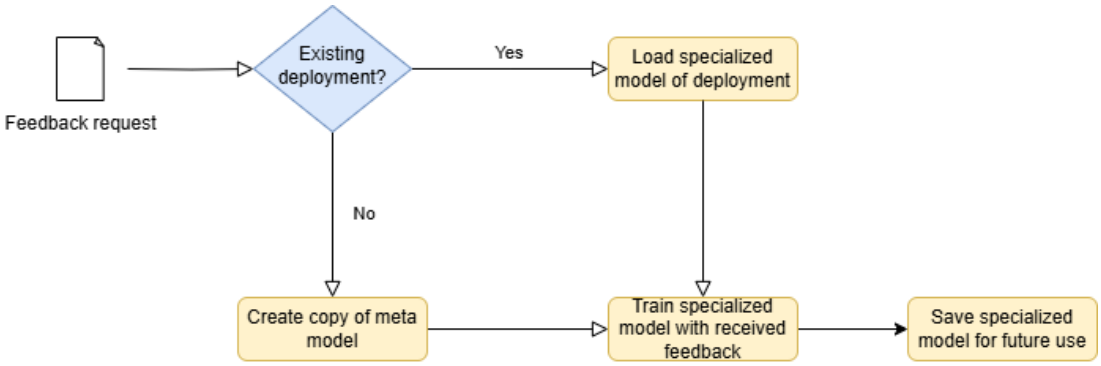

Figure 3.2: Flow diagram of DRL feedback process.

With the specialized models created upon receiving a feedback request, the DRL system takes into account which specialized models are available when new optimization requests arrive. As depicted in Figure 3.3, the DRL system first checks if there is a specialized trained model for that specific deployment and, if so, loads it to use for predicting the optimization step. Otherwise, the meta model is used.

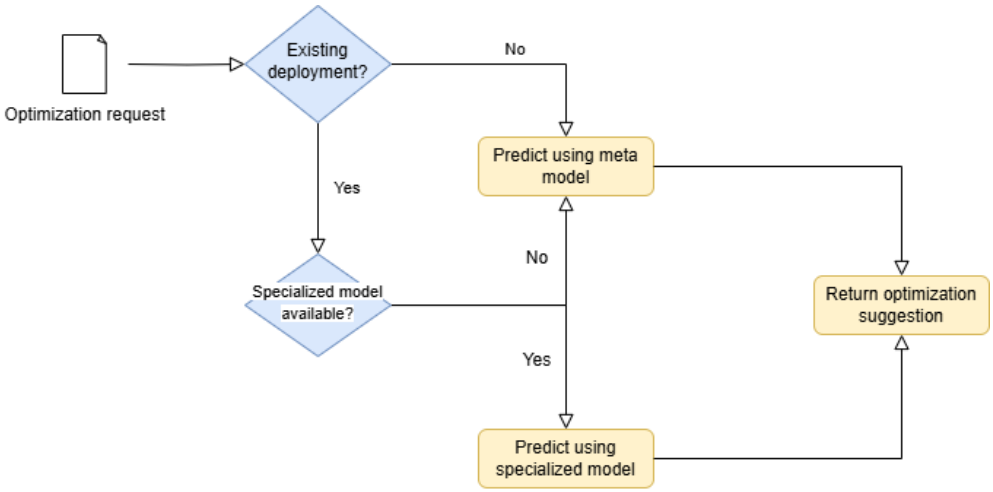

Figure 3.3: Flow diagram of DRL prediction process.

This behavior implies the generation of further deployment-specific DRL models, which

further improve themselves in time in case the same deployment has to be optimized again. The negative feedback is sent by the watcher, which can monitor if a service is performing worse after an optimization suggested by the DRL model. If, for example, the latest latency values are significantly higher than the initial ones for an *"optimization"* prediction, the watcher sends feedback to *"migrate"* instead, in this case, to another node with a reward that scales to the increase in latency.

The following figure describes in more detail the communication between the watcher and the DRL system:

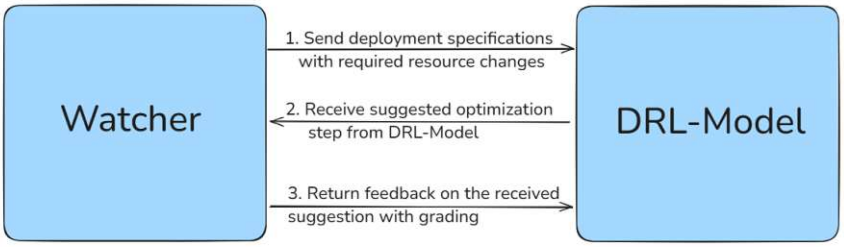

Figure 3.4: Watcher-to-model communication.

After receiving the specification from the agent, the watcher fetches the memory usage and performance metrics from the metrics API and prepares an object with the current state of the deployment and the state that is required by the agent's specification. The watcher sends this object to the DRL system so it can analyze it and return an optimization step based on its trained data, as depicted in Figure 3.4 in step 2. To enable reinforcement learning within the system, the watcher oversees the performance of the affected service. If the performance worsens, the watcher sends the negative feedback to the corresponding DRL model with the correct solution. For this functionality, the watcher needs to keep track of the altered deployments and see how they perform in time after applying the optimization step. Figure 3.5 illustrates the overall DRL system and its instance-based model training layers.

The watcher is directly connected to the DRL System, which can decide on the resource optimization strategy when conflicts are triggered. Based on the watcher's iterative workflow, the model receives the target deployments that are conflicting on each iteration and returns a possible solution based on the provided data. The watcher then applies the action suggested by the model and oversees the performance of the optimized deployment. The watcher sends negative feedback to the model if problems occur, such as latency issues or OOMKills. This feedback system allows the model to improve itself based on its decisions across the iterations. It enables reinforcement learning and can improve the model's accuracy.

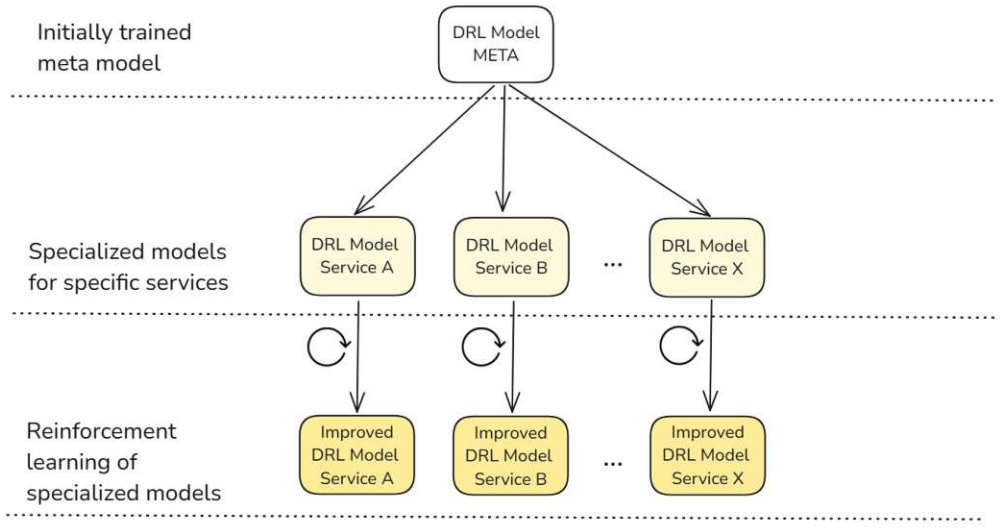

Figure 3.5: Overview of the DRL system.

3.4.6 Overview of Resulting Framework

With this defined framework, it is possible to create a simulation environment consisting of a resource management system and the target resource consumer services to enable us to trigger resource conflicts without difficulty on a high scale, depending on the combination of deployed agents and their specifications. Moreover, in conflicts, the watcher communicates with the metrics API and the DRL system to apply an optimization solution and oversees the evolution of the deployments on which the optimization step was applied. Suppose the optimization leads to errors or worse performance and is not tolerated by the consumer services' resource thresholds. In that case, the watcher sends negative feedback to the specific DRL model used in prediction to help the model improve itself for specific deployment cases.

CHAPTER 4

Implementation

The following aspects have to be taken into account when designing the architecture of the infrastructure where the simulation environment would be deployed:

- Having enough resources available on each host for the main use case of the research. Additionally, having the host set up with similar hardware configurations is necessary for a more accurate evaluation of the results.
- The continuous delivery and integration of the underlying microservices to promote faster development and easy deployment.
- A detailed specification of the components within the environment, which was discussed in the framework chapter.
- Keeping track of the metrics, logs, and performance of the microservices.
- The simplicity of triggering resource conflicts using the agent controller.

4.1 Testbed

The environment for this research will be Kubernetes-based. This technology was chosen because of its vast number of features and reputation. A virtual machine hypervisor will be used to provide the cluster's nodes for more flexibility in the research infrastructure. Harvester, a hypervisor solution by SUSE, is the tool used to create the virtual machines on which the cluster will be built. This VM provisioner is a bare-metal cluster of more servers that allow VMs to be deployed on and use the computation resources[SUS24], as displayed in Figure 4.1.

The virtual machines deployed using Harvester are running the Ubuntu Server 24.04 image and are all provisioned with 300 GB of storage. Each virtual machine automatically

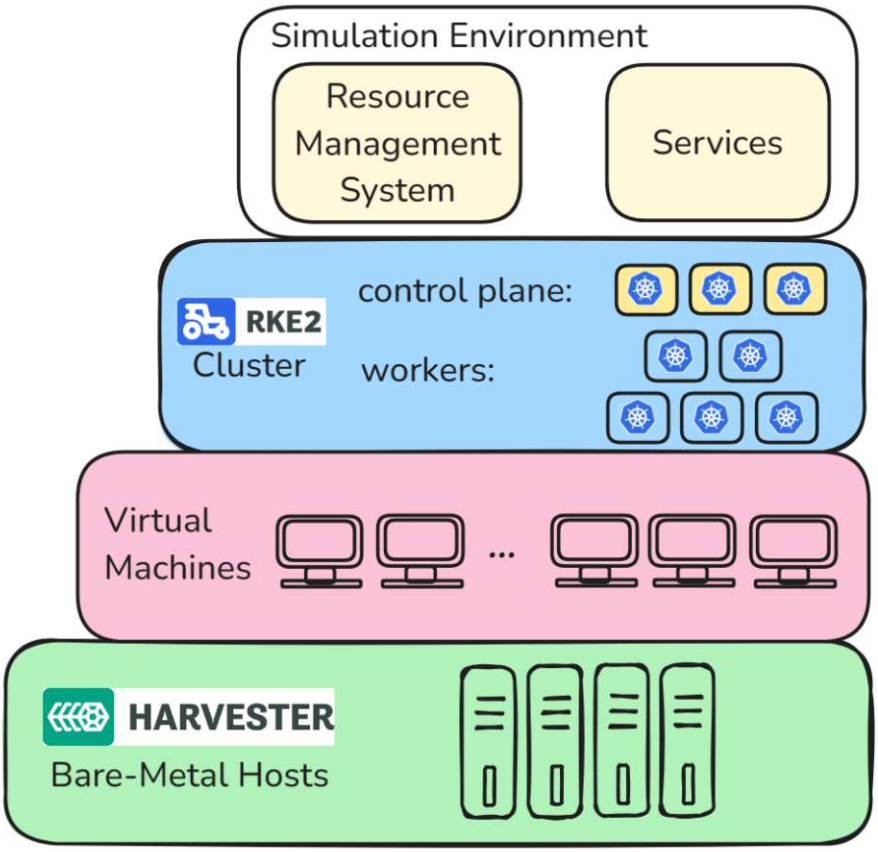

Figure 4.1: Architecture of the system infrastructure.

receives a corresponding IP address from the cluster's subnet with the help of a DHCP server. Regarding the computation resources, the control plane VMs receive 8 CPU cores with 32 GB of memory, whereas the worker nodes receive 4 CPUs with 8 GB of memory. The higher amount of resources allocated for the control plane nodes is required to orchestrate and schedule the Kubernetes cluster and to host the resource management system with all its components. The lower amount of resources was chosen for the workers to simulate a more resource-constrained environment, which may consist of smaller devices with limited resources.

The control plane will be used for managing the cluster itself and deploying the **resource management framework**, whereas the worker nodes will contain the resources to be used by the services. The amount of CPU cores and RAM should allow for a higher playroom when triggering resource conflicts. Four worker nodes have been selected to test the horizontal scaling options of services. This setup will serve as the basis for the resource management optimization tasks of the available deployments, such as vertically or horizontally scaling.

Name	CPU	Memory
Control Plane 1	8 CPUs	32 GB
Control Plane 2	8 CPUs	32 GB
Control Plane 3	8 CPUs	32 GB
Worker 1	4 CPUs	8 GB
Worker 2	4 CPUs	8 GB
Worker 3	4 CPUs	8 GB
Worker 4	4 CPUs	8 GB

Table 4.1: Hardware specifications of the Kubernetes nodes.

Table 4.1 summarizes the cluster's architecture. As can be seen, the K8s cluster will run on the aforementioned hosts, with the control plane nodes also being the core elements of the resource management. The next sections will provide a more detailed overview of each diagram segment.

4.2 Kubernetes' Resource Management

Kubernetes was designed to consider resource allocation when scheduling containers to available nodes, so it includes a resource management system by default. This system provides the functionality for evenly distributing the workloads based on the available resources. However, it is only aware of the resource definitions for the deployments, not their actual resource consumption. This design makes Kubernetes independent of the resource usage metrics because it is only aware of the container's defined resource requests and limits, which can be applied to CPU cores or RAM. The *"requests"* parameter includes the minimum resources reserved for the workload. The *"limits"* parameter represents the maximum amount of resources that can be used by the workload[Ku24].

With these two parameters, Kubernetes can monitor resource allocation as defined in the deployment configurations. Additionally, Kubernetes provides resource management functions that allow the administrator to enforce concrete limits on larger groups of services. Resource quotas are an example of such functionality. They provide constraints that can limit resource usage and allocation within a namespace[Ku24]. Kubernetes does not include resource management functions that target specific nodes only, such as a resource quota that can be applied to a whole node, instead of just a namespace. This feature is included in the framework proposed in this thesis.

The framework does not make use of Kubernetes' resource quota functionality. Instead, it contains its own resource management functions that use the resource request and limit configurations included in the RKE2 cluster. In other words, the framework was designed to be independent and applied to other distributed system technologies, which may provide different resource management functionalities. Therefore, it is important to avoid conflicts between the framework's resource management functionality and the one from the infrastructure in which it was deployed.

4.3 Cluster Service Architecture

The distribution used for the cluster is RKE2. This one has been chosen because of its simplicity in setting up, deploying and managing from a self-hosted Harvester hypervisor. Because RKE2 includes similar functionality as seen in K3S, a lightweight Kubernetes distribution, it also requires fewer resources to run, meaning it can also be deployed on devices using the ARM architecture. It also comes with fewer pre-included services and is flexible when managing its nodes. This feature is ideal because of the constant requirement of restarting the nodes for applying infrastructure changes[rke].

Figure 4.2 shows an overview of the used services in the cluster. Rancher will be used for a management front-end, which is also made by SUSE and mainly supports managing RKE2 clusters. The service networking will be handled by MetalLB, which excels in providing external IP addresses to services running in the cluster. A Prometheus and Grafana stack, which is displayed in the figure, will be deployed on the cluster to enable functionality and the ability to monitor resource usage. A Jupyter Lab instance is also deployed on the cluster for feature and functionality development and testing. The simulation environment will run across the nodes, with the resource consumer services being deployed on the worker nodes.

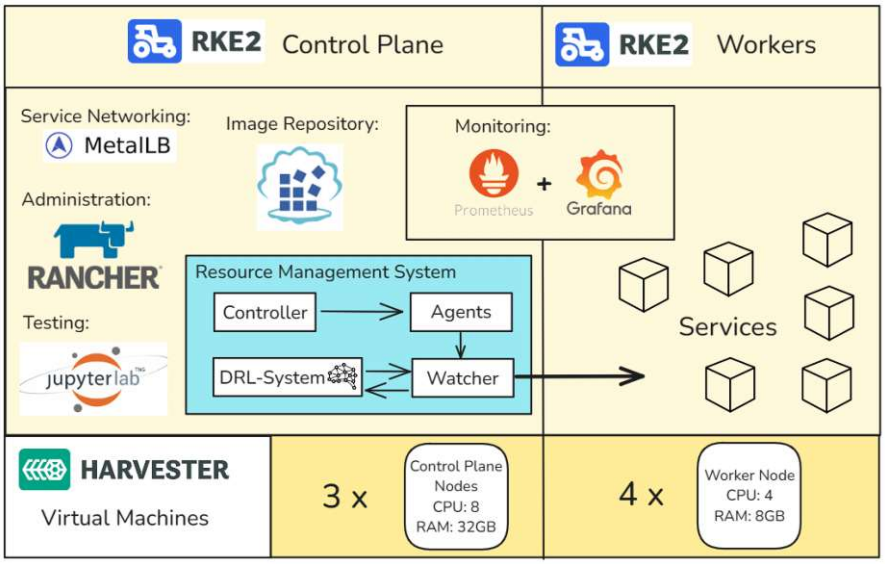

Figure 4.2: Cluster architecture diagram.

4.3.1 Networking

Because this cluster runs in bare metal within a subnet, it is not possible to use a network load balancer without further plugins since the cluster does not run on a supported IaaS platform. MetalLB solves this issue with its Kubernetes operator by using a configured object that describes the IP pool of the available addresses to be used by the services

and assigns them to created services of type LoadBalancer[Met]. This allows the services to be reached using an external IP.

4.3.2 Monitoring

As mentioned in the previous sections, a monitoring system will be required, which tracks the performance and latency of the services that run within the environment. Prometheus will be used in combination with Grafana for monitoring the cluster. While the Prometheus API will be useful for providing the data required when training the DRL model for conflict-solving, the Grafana dashboards will grant insight into the workloads within the clusters [gra]. An overview of the services running on the worker nodes is highly important because of the limited resources available.

Aside from the normal setup of the monitoring stack, additional settings had to be configured for both Prometheus and Grafana. One of the most important changes was enabling the persistence of the data and having it save the metrics for up to thirty days. Additionally, this implies that the custom dashboard in Grafana that was created for this research also remains persistent between more redeployments of the Grafana service.

On the Prometheus side, a special job was configured that creates dynamic target endpoints only for resource-consumer services that run in the simulation environment. With additional configurations on the consumer side, it is possible to enable Prometheus to scrape performance metrics from the containers, such as the latency of a function.

Grafana, on the other hand, requires additional configurations to allow dashboard embedding on other websites. This is an important feature for the custom dashboard that is included in the agent controller. To enable embedding, *allow_embedding=true* has to be included in the *grafana.ini* configuration.

4.3.3 Private Docker Registry

For deploying own images of the consumer services, agent controller, watcher, and DRL system, a private registry was set up which runs inside the cluster on the control plane and is reachable by the load balancer IP address obtained from MetalLB[lea]. This Distribution registry is provided by the Cloud Native Computing Foundation[CNC25].

4.3.4 Jupyter Lab

Jupyter Lab is an additional tool that simplifies the feature development and testing of the complex algorithms of the containers. It is an open-source computing environment that was first developed to support Python kernels, which can be used to [jup]. Because the watcher's code relies heavily on the Kubernetes Python package and the API, testing functions within the Jupyter instance was simpler because of its in-cluster configuration. This implies that a service account was created for Jupyter Lab to be able to grant it permissions for making the required Kubernetes API requests.

4.4 Resource Conflict Simulation Environment

With the main architecture of the cluster up and running, it is now possible to focus on the main containers that will interact with each other to simulate the research's use case and trigger conflicts between resource requests. A goal is also to make this system as independent as possible. Thus, it will not use any Kubernetes-included plugins or extensions made by the community that are used in resource management.

Figure 4.3 shows the container structure of the simulation environment, which first consists of the resource management system which runs on the control plane. Next are the resource consumer services, which are containers that run on the worker nodes. On the resource management side, there is the agent controller, which receives input from the administrator and uses it to create the agents. After consulting and receiving a suggestion from the DRL system, the watcher oversees the state of the agents and applies their specifications to the targeted services, depending on the scope that the agent has been configured with. The possible scopes of resource management are depicted with numbered arrows. In case 1., the watcher applies a resource quota on a whole node. Case 2 illustrates an agent with a specification requesting a resource quota for a namespace. Case 3 depicts the case where the agent specification targets only a single service deployment. In addition, the communication with the metrics API is also shown.

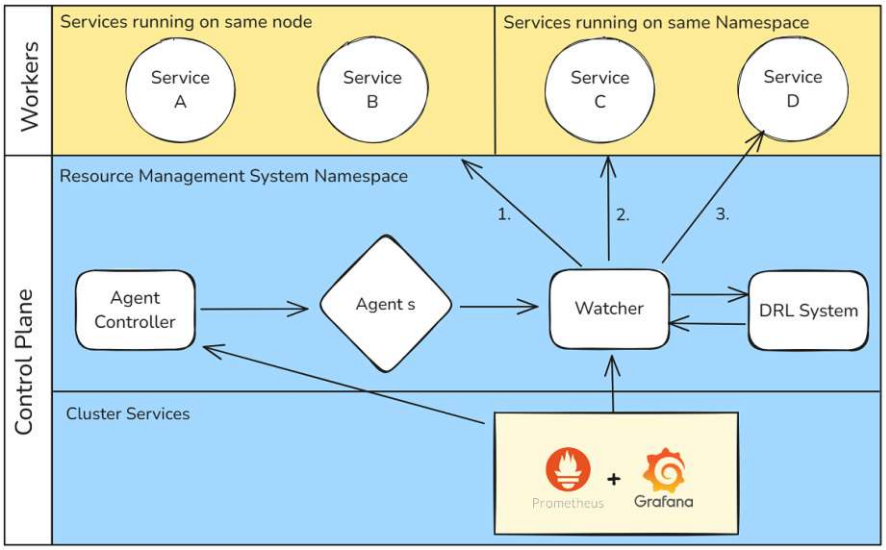

Figure 4.3: Overview of the simulation environment in the RKE2 cluster.

4.4.1 Consumer Services

The consumer services are containers that run as deployments in the Kubernetes cluster on the worker nodes. They represent the services that have to run in the computing continuum and be optimized by the resource management system. A container runs a

simple Python script that trains a model repeatedly, keeping track of the time required to train the model. This time is used as a metric to measure the performance of the service and is sent to the Prometheus instance. Prometheus was configured to include a dynamic endpoint targeting consumer services for this functionality. This allows Prometheus to scrape the metrics gathered in the containers and monitor and feed the data to the watcher.

A very important aspect of this implementation is Kubernetes' deployment strategy parameter. To enable computing a continuum for consumer services, the strategy parameter needs to be set to *"RollingUpdate"*, which does not allow downtime of the container. With the help of a readiness probe, it is possible to make the deployment object aware of the start of the workflow in the newly created pod. When the workflow within the new pod is running and the readiness probe detects it, the deployment may delete the old pod and finalize the optimization procedure. This ensures that the service workflow is continuous.

4.4.2 Management Components

The management components include the agent controller, the agents, the watcher, and the DRL system. All these components, except the agents, are deployed on the control plane node since they require more computation power for the management system. These deployments are all reachable through their service objects created in Kubernetes and can communicate with each other. Conversely, the agents are a Custom Resource Definition (CRD) within the RKE2 cluster.

4.4.3 Agent

Kubernetes allows the creation of custom resource definitions (CRD), which can be seen as extensions of the Kubernetes API itself. They work as endpoints in the API and store objects of specific structures, which can be defined in a YAML configuration [Ku24]. This customization is of high importance in this research because CRDs can represent the state of the agents created by the controller without needing to set up a database that keeps track of the agent request states.

The agent is a CRD created specifically for this use case, allowing the controller to create or edit the agents in the Kubernetes state. Having the agent designed as a CRD makes it centralized and simplifies the communication between the controller, the watcher, and the DRL system. The scope parameter changes the goal of the CRD completely, for either applying the rule on a specific deployment, namespace, or a whole node.

This implementation is highly important because it allows saving the agents with their configurations and statuses within the cluster's state. This also implies that this information would in turn be reachable by the other components of the management system through the Kube API. Depending on their status, having the agents saved in the state allows the watcher to observe them and apply their specifications to the cluster.

4.4.4 Agent Controller

The agent controller is deployed as a Python container with the FastAPI and Kubernetes packages, which support its main functionality. Because the controller needs the permissions to create the agent objects within the cluster, a ServiceAccount object must be created that would be bound to a cluster role. This cluster role grants the agent controller the required permissions to create and manage agents within the Kube API.

Controller Dashboard

Aside from its main functionality discussed in the framework section, the agent controller also serves as a centralized control panel for the whole resource management system. Using FastAPI, the controller system provides a frontend user interface that serves not only as a dashboard but also as an interface for creating and managing agents in resource management. The following screenshots show the layout of the dashboard. Figure 4.4 shows the form that needs to be filled out to configure the resource management agent. Figure 4.5 shows the overview of the main page in the dashboard. The agents tab can be seen in the upper section, along with the available agents. The deployment overview shows the deployments running in the simulation environment on the four nodes. It can be observed that for the first three deployments, the action *"0"*, which means *"optimize"* was selected. This means that these deployments are currently optimized and managed by a node-wide agent with a memory resource quota of 90%. The applied memory based on the aforementioned quota can also be seen in the figure. Figures 4.6 and 4.7 show the monitoring tab of the dashboard, which includes iframes tailored to oversee the resource consumption and the latency of the services within the simulation environment.

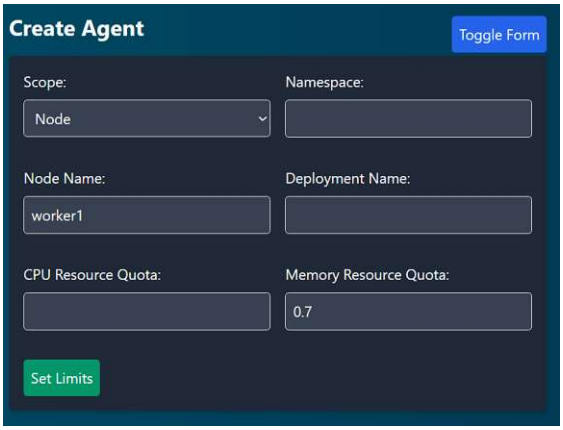

Figure 4.4: Overview of the agent creation form.

LLM-powered, Prompt-based User Inputs

The agent controller also offers an LLM-powered prompt processing system. By integrating LLM functionality, the controller can intelligently process and analyze incoming user

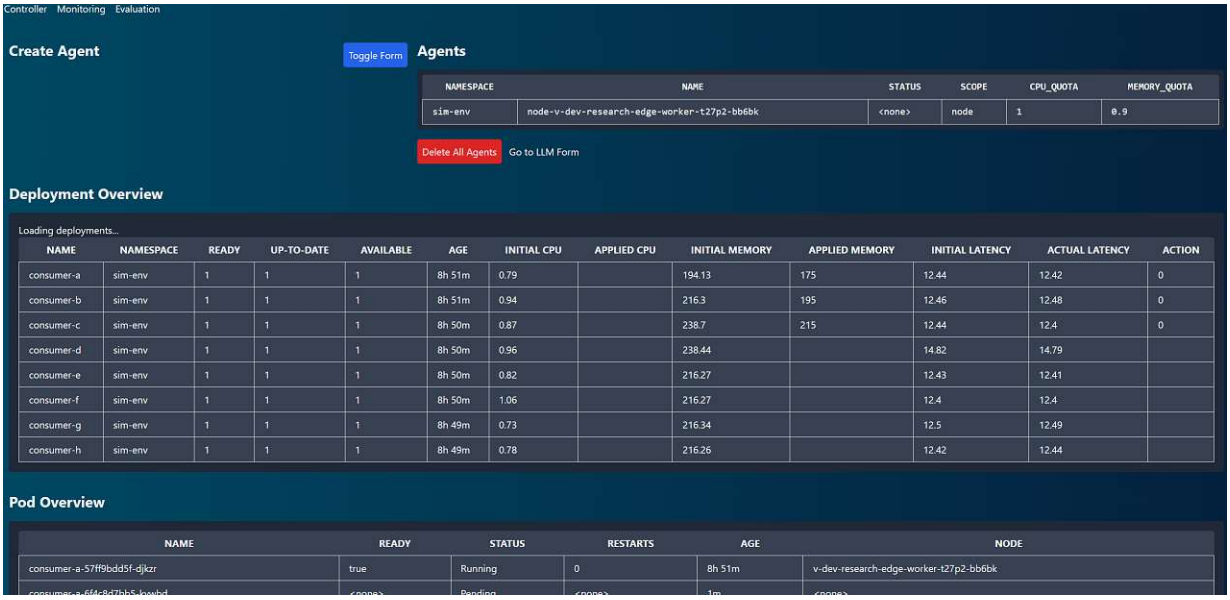

Figure 4.5: Dashboard main view.

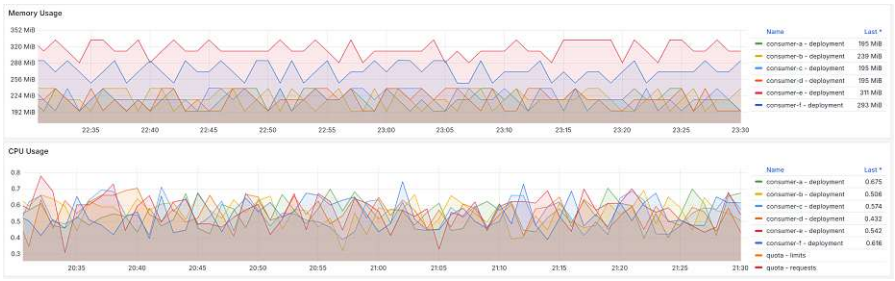

Figure 4.6: Overview of the CPU and memory usage graphs seen in the dashboard.

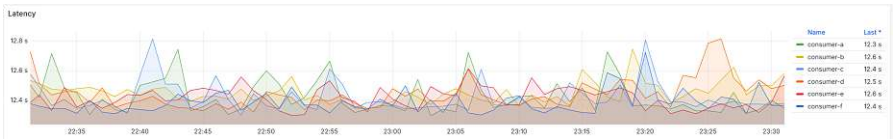

Figure 4.7: Overview of the latency graph seen in the dashboard.

orders, interpret context-specific resource requirements, and generate agent specifications with increased precision and adaptability. These specifications would then create the agent specification using the correct syntax.

The agent controller supports an API connection to OpenAI, allowing it to create an instance with the available models. For this implementation, the model *gpt-4o-mini* is used. This LLM has been chosen because of its smaller size and cost efficiency. It is also smarter and cheaper than the *GPT-3.5 Turbo* [Ope24]. The OpenAI API has a

huge advantage in configuring the model it connects to. The main goal for this model is to receive the input, which would be in normal text structure, and map the content, if relevant, to a pre-defined object structure that is set for the model.

```
    {"role": "system", "content": "Convert user input into a JSON dictionary
        with this structure: \n\n"
                          "\"spec\": {\n"
                          "   \"scope\": string (specified scope),\n"
                          "   \"nodeName\": string (if specified) or \"\",\n"
                          "   \"namespace\": string (specified namespace) or
                              \"\",\n"
                          "   \"deploymentName\": string (if specified) or
                              \"\",\n"
                          "   \"resources\": {\n"
                          "      \"resourceQuota\": {\n"
                          "         \"cpu\": float (if specified) or 1.0,\n"
                          "         \"memory\": float (if specified) or 1.0\n"
                          "      }\n"
                          "   }\n"
                          "}\n\n"
                          "Ensure to parse the input command and extract
                              details into the dictionary accurately. Use empty
                              string \"\" instead of None for values that are
                              not specified."},
{"role": "user", "content": prompt}
```

Figure 4.8: LLM configuration for parsing user input.

As can be seen in Figure 4.8, it is possible to specify the role of the user and the model's role. By specifying the structure of the object and the values that are expected to be used for each key, the OpenAI model can process the input for the desired use case. An additional rule is mentioned for handling missing values and avoiding using *"None"*, where missing values should be empty strings instead.

4.4.5 Watcher

The watcher is one of the more complex implementations in this system because of its many tasks that have to take place iteratively so that the occurring events in the cluster, such as the creation of an agent or the change of its state, can be covered and as quickly perceived as possible. The workflow of the watcher consists of the following points, where the indentations represent inner loops in the code:

1. Initialization of in-memory data frame, configuration of communication endpoints, and authentication to the Kube API.

2. The *"watch"* function, which iterates every five seconds. The value can be configured to change the frequency of the iterations. This function keeps the data frame up to date with the relevant deployments targeted by agents. On this high-level iteration,

the watcher makes sure that relevant agents still manage the deployments. In case an agent has been deleted, the targeted deployments are reverted to their normal configuration before the agent's changes. Additionally, the watcher sends information from this data frame to the controller dashboard for tracking and monitoring purposes.

- After verifying that the deployment-agent information is up-to-date, the watcher checks every agent in turn and, based on its current state, either applies it or checks it.
- Applying an agent's specification is another complex function, which first checks the agent's scope, obtains the target deployments, requests their latest metrics for the resource usage and latency, and applies the agent's required resource quota on their actual resource usage. In this step, the communication with the DRL system occurs.

The watcher also includes a thorough logging system that outputs the decisions it takes regarding the various states of the deployments and agents per iteration and its communication with the DRL system and the controller component.

4.4.6 DRL System

The DRL system is the container that manages all the trained DRL models and allocates them to the corresponding deployments they specialize in, upon receiving a prediction request. As a standard to be used by the reinforcement learning task, the following section describes the training process of the meta DRL model.

DRL Model

The DRL model is a feed-forward, fully connected neural network consisting first and foremost of the input layer, which is used to cover the shape of the entries when introducing them in the model[ZTB+18]. Afterwards, the model consists of three hidden dense layers, which capture the low-level patterns between the features before further compressing the learned features and preparing them for the final output layer. The first hidden layer consists of 64 neurons, followed by the second with 32 neurons and the third with 16. Stacking layers makes learning complex decision boundaries and patterns easier for the model. Dropout layers have been added between the dense layers to promote learning and reduce the risk of overfitting. These layers randomly disable neurons during training to prevent memorization[PK17]. The output layer has 4 neurons, each corresponding to one class. This implies that the neural network outputs a probability distribution across all four labels used in training. An embedding layer is included in the model to make the neural network aware that the training data is not related to a single deployment only and that it should encompass a collection of deployment entries with various threshold ranges. This layer allows the neural network to differentiate between unique cases, essential in reinforcement learning. The neural network structure can be seen in Figure 4.9.

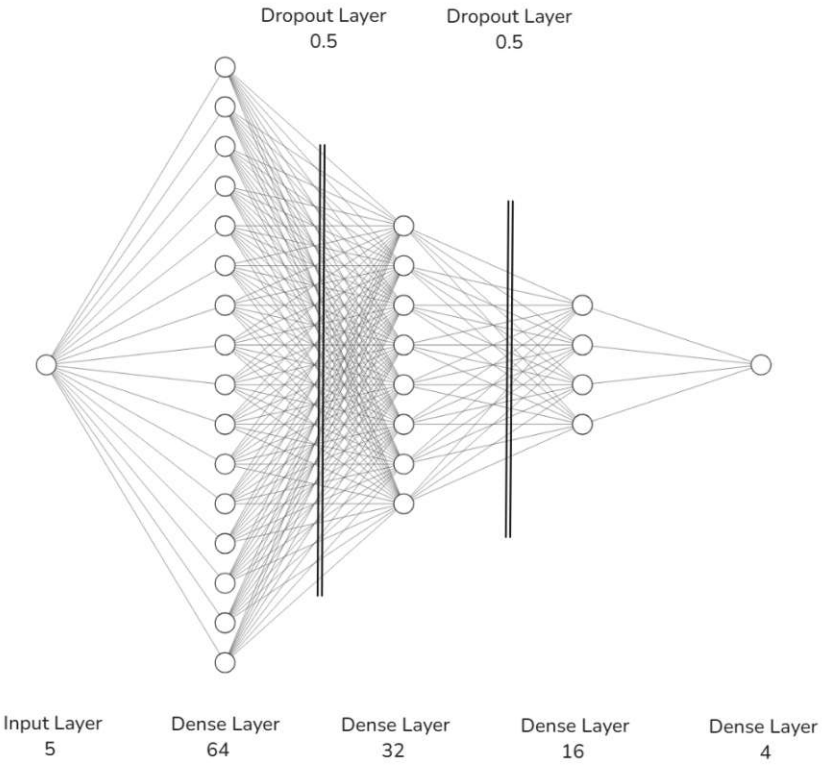

Figure 4.9: Architecture of neural network's model.

To first train the model with the logic basis of the use case, 12000 entries have been automatically generated. Subsets of these entries contain various configurations for covering the possibilities that may lead the deployment under specific conditions to the corresponding label. Algorithm 4.1 shows an overview of how the training data is generated for the neural network model. This algorithm ensures that the main cases of optimization are equally distributed and cover the average resource thresholds that a normal container would have. Regarding the latency category, it is created to teach the model to prioritize the latency when making a decision in case the performance of a service has decreased significantly.

Algorithm 4.2 describes the model training process in detail. The first six steps encompass the pre-processing stage of the workflow. The algorithm ensures that the values in the data set are standardized. This improves the quality of the model because the neural network used performs better when input features are on a similar scale[ZTB+18]. TensorFlow and Keras are used to train and test the model with the support of a consumer-grade GPU. The model training phase takes around one minute.

Table 4.2 displays the hyperparameters used in training the DRL model. The optimizer used is Adam with a 0.001 learning rate. The value of the batch size is equal to 16.

Hyperparameter	**Value**
Optimizer	Adam
Learning Rate	0.001
Batch Size	16
Epochs	100
Activation Functions	ReLU (hidden), Softmax (output)
Loss Function	Sparse Categorical Crossentropy
Dropout Rate	0.5
Hidden Layers	3
Units per Layer	62, 32, 16
Callbacks	EarlyStopping, ReduceLROnPlateau

Table 4.2: The hyperparameters used when training the neural network.

This value decreases a bit the processing speed on each epoch, however it helps increase the generalization capabilities of the model. The activation functions are ReLU for the hidden layers and Softmax for the output layer. The loss function chosen for this neural network is the sparse categorical cross-entropy. It was used because of its performance in multi-class classification and for supporting labels which are not hot-encoded[Cha23]. With the aforementioned train set, significant performance has been reached in at least 20 epochs with a configured early stopping. Until that number of epochs, the model was still not able to learn and generalize accurately with the available data. Early stopping was used to stop the model before a decrease in performance after a specific number of epochs. With the optimal training configuration, the neural network reaches an accuracy of 97.5%. The plots 4.10 and 4.11 show the loss and accuracy changes over the epochs when training the model. In this state, the trained neural network, upon receiving the features of an actual deployment, can infer by returning a probability distribution of the four labels.

After performance tests, the model is saved and deployed in a container using a Python script that loads the required files and uses fastAPI, which provides two endpoints that are used to interact with the trained model: *"predict"* and *"feedback"*. The watcher uses these endpoints to send deployment information and receive the suggested optimization step from the model. Afterward, the watcher gives feedback on the suggested prediction. To be able to track the predictions and the feedback responses, the deployment IDs are linked to the JSON objects sent between the resource management services.

By dynamically creating and further training specialized models for the deployments that are not optimized well with the meta DRL model, it is possible to achieve more accurate predictions for these specific cases, as there is also a history of prediction requests for the specific deployments. Returning negative feedback as soon as the watcher observes performance issues helps the DRL system to improve itself quickly and be aware of the stricter resource thresholds of these deployments if a future need for optimization arises. This approach was selected for this DRL system because further training the

Algorithm 4.1: Train Data Generation for Optimization Tasks

Ensure: Generate training data for optimization tasks

```
Define optimization tasks: [optimize, migrate, scale, latency]
for each task in optimization tasks do
    Initialize empty list for task data
    for each of the 3000 entries do
        Determine which third of the dataset the entry belongs to
        if entry belongs to the first third then
            Set memory quota thresholds
            Trigger optimization task based on memory quota
        else if entry belongs to the second third then
            Set CPU quota thresholds
            Trigger optimization task based on CPU quota
        else
            Set both memory and CPU quota thresholds
            Trigger optimization task based on both quotas
        end if
        Add entry to task data list
    end for
    Save generated task data for the current task
end for
```

Algorithm 4.2: Model Training

Ensure: Train and save the DRL model based on deployment data

```
Load and pre-process training data
Select features and labels
Apply StandardScaler to scale numeric features
Split dataset into 80% training and 20% testing sets
Save the fitted scaler as backup
Define the model architecture using dense and dropout layers
Set model hyperparameters (learning rate, loss function, optimizer)
Train the model using training data
Evaluate model performance on testing data
if performance meets required threshold then
    Save the trained model
end if
```

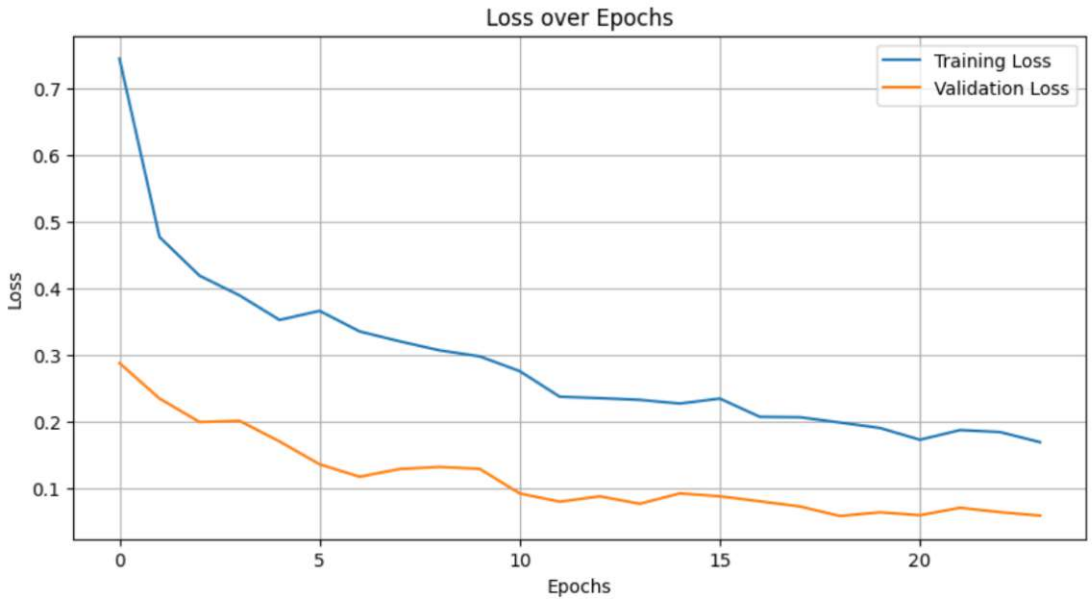

Figure 4.10: Progression of loss during training.

main meta model with thresholds that would violate its trained knowledge would lead to generalization and significant changes in decision weights [GLU20]. Additionally, Q-learning would have made little sense in this use case because it is more fit for decision chaining[LLX+17].

Reinforcement Learning Module

After training the model, it can be saved and then loaded into the DRL system, which provides two endpoints: *"predict"* and *"feedback"*.

The first endpoint receives the deployment with the body structure as used in the model training, with the actual values observed in the simulation environment. It feeds the object to the model corresponding to the deployment ID and returns the optimization step suggested by the neural network.

The second endpoint receives another object that includes the feedback of previous predictions made by the model. The connection between the feedback to the previous prediction may be done using a unique identifier. The body of the received feedback object contains the unique identifier of the deployment, the correct optimization step, and a rating, which may be interpreted as the impact of the predicted choice. This information is used to train a specialized model that will be used for future predictions for this specific deployment. Using this instance-based technique, it is possible to tailor a clone of the main DRL model to distinguish for this deployment's specific resource requirements. This self-correcting mechanism allows the overall DRL system to adapt to the specific needs of the deployments. Because of the online feedback, the DRL System

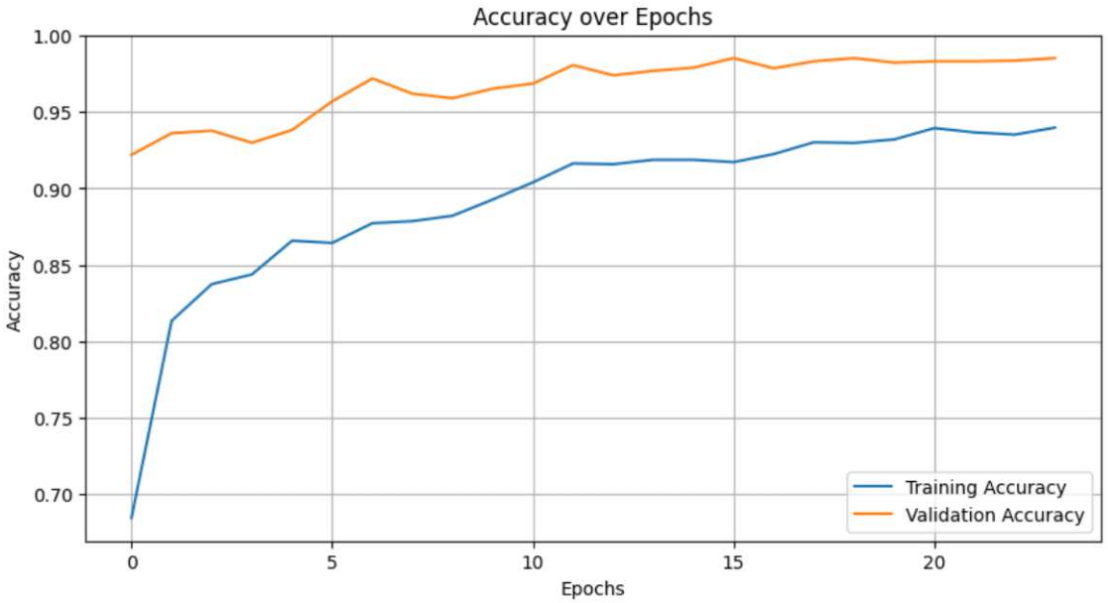

Figure 4.11: Accuracy over epochs.

should also be able to adapt to changing workloads, in case unpredictable changes occur.

4.4.7 Detailed Overview on the Finalized Workflow

The source code of our implementation is pushed in the *"DRL-RMS-Framework"* repository[PV25]. With the framework applied in the architecture of the simulation environment, it is now possible to generate agents with specific resource management tasks and have them applied by the watcher. This also implies the possibility of generating conflicts within this environment. We distinguish two types of conflicts that may appear in this simulation: conflicting agent specifications and conflicts in optimization.

4.4.8 Conflicts of Agent Specifications

Specification conflicts are to be distinguished from the conflicts that may occur after applying an agent's specification. Internal logical conflicts are triggered when an agent's specification contradicts another. The following example should clarify such a condition:

1. Some microservices run in the same namespace across more nodes.
2. Agent A is created to reduce the CPU consumption of all microservices in a namespace by 30%.
3. Because of a sudden need to increase the CPU usage of one of those microservices, agent B is created to increase the CPU usage of microservice B by 50%.

4. The watcher applies the agents' specifications in order, and because agent A keeps decreasing the CPU. In contrast, agent B increases it; both agents end up in an endless loop overwriting each other.

While these conflicts are not as complex as the conflicts in resource optimization, they may become more difficult to solve if the number of deployed agents with conflicting specifications increases.

For being able to oversee the specification conflicts of the agents, the watcher uses a system that counts the times an agent has been reapplied by keeping count of the iterations in the watcher loop. By using the agent's specification, the *"restarts"* parameter keeps track of how many times it has been reapplied. Because the watcher is aware of its iteration counts, it is possible to save the restart count of the agent and the iteration count at runtime as a dictionary for each agent. If the condition in which the restart count for an agent increases by one on each iteration is fulfilled, the watcher flags the agent as "conflicting". Additionally, because the restart counts always increase for all the conflicting specifications, the watcher can detect all the conflicting agents. If the watcher observes this behaviour on each iteration, the latest agent that triggered the conflict is labelled accordingly with the *"disabled"* flag and no longer reapplied to the environment. Upon using this method, the administrator is notified of the existing conflict, so an action may be taken.

4.4.9 Conflicts in Resource Optimization

After applying an agent's specification, an uninstructed watcher would simply apply the rules without considering the impact these resource changes would have on the target services' performance. This may lead to problems within the cluster and even to service downtime. For example, setting a rule to decrease the memory consumption in a node by 30% might work for some services, but if there are services that need those resources they would either perform worse with a higher latency and longer processing time or an out-of-memory kill (OOMKill) would be reached. Once this state is reached, the service container would keep restarting itself, however, with the same amount of resources.

Being able to roll back to their previous well-performing configuration is of high importance, as well as the ability to migrate those failing services to other nodes, where more resources would be available. These are functionalities that the watcher covers. However, the decision to take these actions does not rely on the watcher, but on the deep learning model developed for this research.

CHAPTER 5

Benchmarking and Evaluation

The framework benchmarking will occur on the same infrastructure documented in the implementation chapter. Benchmarking will be made for each of the aforementioned conflict types. In this section, we not only evaluate the speed of the conflict-solving and the accuracy of the implemented DRL system but also the quality of maintaining a workflow when optimizing its resources to promote the computing continuum.

5.1 Benchmarking Conflict-Solving in Agent Specifications

This benchmark depicts a conflict where two agent specifications contradict each other and tests how the implemented framework solves such events.

Initial State

Three agents are to be found in this environment, agent A for the whole *namespace-1* which has already been applied, and two pending agents. Agent B wishes to increase the memory limit only for the deployment *consumer-app*, which runs on *namespace-1*. Agent C applies a resource quota on a different namespace. Therefore, agents B and C are both in the pending state and waiting to be implemented by the watcher. This state is displayed in Table 5.1.

Name	State	Scope	Scope Target	Memory Limit
Agent A	Applied	Namespace	namespace-1	6Gi
Agent B	Pending	Deployment	consumer-app	8Gi
Agent C	Pending	Namespace	namespace-2	12Gi

Table 5.1: Initial state of the deployed agents.

Conflict Detection

With the help of the iteration system described in the framework, the watcher can track the number of consecutive times the agents have had to be reapplied. In such an occurrence, the conflicting agents are marked as *"conflicting"* until the agent that triggered the conflict is disabled from reapplying, as seen in Table 5.2.

Name	State	Scope	Scope Target	Memory Limit
Agent A	Conflicting	Namespace	namespace-1	6Gi
Agent B	Conflicting	Deployment	consumer-app	8Gi
Agent C	Applied	Namespace	namespace-2	12Gi

Table 5.2: Agents A and B are set to conflict.

Agent B is overwriting the memory quota imposed by agent A and vice versa. The agents remain in conflicting state until the latest agent that triggered the conflict is disabled, as seen in Table 5.3. Agent B remains in the *"disabled"* state until the administrator resolves the configuration error. Agent C switches to the applied state because it does not conflict with the other agents' specifications.

Name	State	Scope	Scope Target	Memory Limit
Agent A	Applied	Namespace	namespace-1	6Gi
Agent B	Disabled	Deployment	consumer-app	8Gi
Agent C	Applied	Namespace	namespace-2	12Gi

Table 5.3: Agent B is disabled to prevent the conflict.

Evaluation

The watcher can detect conflicting agents that must be reapplied on each iteration, and this system scales well with more agents. However, it does not point out the specific configuration problems that have to be changed by the administrators.

The time required for the watcher to detect an agent configuration conflict is around 15 seconds, with a 5-second set per iteration. The main reason for this is the fact that the watcher observes if the agents have been applied at least three times in a row and then flags them as conflicting. In all test cases, the algorithm detected the last applied agent that triggered the conflict chain and disabled it.

5.2 Benchmarking Reinforcement Learning of DRL System

Before evaluating the conflict solving capability of the resource management system, it is important to benchmark and assign a performance rating to the reinforcement learning

feature included in the DRL system. To simplify this test case, we apply an agent with a specific quota on a node with a single service running on it, so that the DRL system can suggest a decision that would lead this service to a worse performance or even downtime.

Test #	Replicas	Initial Memory (MiB)	Memory Quota (%)	Performance (%)	1. Prediction	Correction	2. Prediction
1	1	215	40	0	optimize	migrate	migrate
2	2	426	75	70	optimize	scale down	scale down
3	1	531	80	60	optimize	migrate	migrate
4	4	990	70	55	optimize	scale down	scale down
5	1	478	65	50	optimize	migrate	migrate
6	1	410	90	72	optimize	migrate	migrate

Table 5.4: Test cases for benchmarking the reinforcement learning module.

Evaluation

As can be seen from these test cases in Table 5.4, the specialized DRL models created specifically for these deployments remember the correct choice when facing a similar optimization request for that specific deployment. The test cases selected here are cases that do not correspond to the meta DRL Model's thresholds and are therefore a great example of showing the learning factor in application. For example, test case 1# reaches an OOMKill state after the DRL systems decide to reduce its usage with this memory reduction quota. The watcher, in this case, would detect the bad performance and revert the service to its original state and mark it as *"high priority"*. On a fresh reset to the initial state, the quota is again applied, and the DRL system, now with the learned insight, suggests migrating the service in test 1# to another node.

Additionally, test case 6# shows only a small memory reduction of 10% of the initial memory usage of 410 MiB of the service. This service, in turn, reacted more drastically after the RAM reduction, and its performance lowered by almost 30%, which would violate its SLOs. Because of this service's higher resource needs, the model is trained to migrate this service instead and flag it as *"high priority"*.

With this information in mind, we can proceed to benchmark the whole resource management system to solve conflicts in resource optimization.

5.3 Benchmarking Conflict-Solving in Resource Optimization

The main goal of benchmarking resource optimization is to show how the resource management system performs when applying a resource quota on a node-wide scale. In practice, without such a system, this process would take a long time to apply and may lead to many service failures. This is why this benchmarking aims to see how quickly the node-wide services are optimized based on the set quota, how the DRL System reacts to conflicts, and how it learns to avoid them. Benchmarking will be done on the four worker nodes mentioned in the implementation chapter.

The following benchmarks are made in environments with similar services running on the nodes and environments with applications with different specifications and resource consumption patterns. The distribution of the services across the nodes can be seen in Table 5.5. Some configurations, such as the number of replicas or resource usage, will be changed to be able to simulate specific cases that require different approaches made by the resource management system. All benchmarks have been done multiple times to detect differences in the DRL model's decision-making and to oversee its learning capabilities for future cases of the same deployments, in case wrong predictions have been made.

Deployment Name	Replicas	Node
consumer-a	1	worker-1
consumer-b	1	worker-1
consumer-c	1	worker-1
consumer-d	1	worker-2
consumer-e	1	worker-2
consumer-f	1	worker-3
consumer-g	1	worker-3
consumer-h	1	worker-4
consumer-i	1	worker-4

Table 5.5: Distribution of the deployments across the nodes.

5.3.1 Benchmark: Node-wide Resource Optimization

This benchmark tests how the framework performs when applying a node-wide resource reduction, how well the usage optimization is done and how conflicts in optimization are solved.

Initial State

Ten similarly configured consumer deployments are running on the four worker nodes that have already been running for some time, so that Prometheus can obtain and process their resource usage and performance. By default, the services are scheduled to the nodes so that the available resources are allocated equally, so there would not be any nodes with a higher workload when nodes with more free resources are available. However, in this case, we have more services that must run on *worker-1* and would therefore be interesting to see if we can reduce the costs of that node by setting an agent that would decrease the CPU usage by 30%.

The following Table 5.6 shows the initial state of all running deployments:

Table 5.7 shows the initial resource usage and allocation for the worker nodes. Please take note of the difference between the total reserved memory and the actual usage. The reserved memory is so high because each consumer service was configured with a resource request of 1512 MiB, from which only around 230 MiB are used. The CPU usage always

Deployment Name	Replicas	Node	Initial CPU (cores)	Initial Memory (MB)	Initial Latency (s)
consumer-a	1	worker-1	1	230	12.6
consumer-b	1	worker-1	1	230	12.2
consumer-c	1	worker-1	1	230	12.1
consumer-d	1	worker-2	1	200	12.5
consumer-e	1	worker-2	1	230	12.4
consumer-f	1	worker-3	1	215	12.6
consumer-g	1	worker-3	1	210	12.6
consumer-h	1	worker-4	1	230	12.6
consumer-i	1	worker-4	1	215	12.6

Table 5.6: Starting state of all deployments with initial resource allocation and latency.

Node	# of Pods	Total CPU Usage (%)	Total Reserved CPU (%)	Total Memory Usage (%)	Total Reserved Memory (%)
worker-1	3	75	75	0.86	56
worker-2	2	50	50	0.57	37
worker-3	2	50	50	0.57	37
worker-4	2	50	50	0.57	37

Table 5.7: Node resource overview with pod count, usage, and reserved resources.

fluctuates during the workflow, which is why this table shows the configured number of cores.

Agent Creation

Using the controller, we create an agent with the following specification:

- **Scope**: *"node"*
- **Target Node**: *"worker1"*
- **CPU Quota**: *0.85* (Quota of 85% implies a decrease of 15%)

Optimization

As soon as the watcher detects the newly created agent, it obtains the objects of the targeted deployments using the Kubernetes API, fetches the initial CPU usage as documented in the column of Table 5.6, and applies the CPU quota of 30% of the actual CPU usage.

Observation

Table 5.8 depicts the resulting states of the deployments after being optimized by the watcher and redeployed with the new CPU specifications.

As seen in the changes on the node level in Table 5.9, the total CPU usage has been reduced by almost 15%, as well as the reserved CPU on the node.

Deployment Name	Replicas	Node	Init CPU	Applied CPU	Init Lat.	Latest Lat.	Action
consumer-a	1	worker-1	0.94	0.80	12.6	14.9	optimize
consumer-b	1	worker-1	0.92	0.78	12.2	15.1	optimize
consumer-c	1	worker-1	1	0.85	12.1	14.3	optimize

Table 5.8: Optimization results after applying the CPU usage quota.

Node	# of Pods	Total CPU Usage (%)	Total Reserved CPU (%)	Total Memory Usage (%)	Total Reserved Memory (%)
worker-1	3	60.75	60.75	0.2	56

Table 5.9: Results of the applied CPU quota on the node level.

Additional Tests with Different Quotas

The following Table 5.10 shows test results where different combinations of quotas have been applied to the same services starting from the same initial state, some cases including triggered conflicts.

Re-test #	CPU Quota (%)	Memory Quota (%)	Avg. Performance (%)	Conflict Triggered	Conflict Solved
1	60	100	62.5	Yes, by optimizing	Yes, by migration
2	100	60	0.0	Yes, by optimizing	Yes, by migration
3	90	80	93.1	No, by optimizing	-
4	100	50	100*	No, by migration	-

Table 5.10: Results of additional tests made with different resource quota combinations.

* The average performance 100% for test case 4# remains equal because the service was migrated to another node before the occurrence of a conflict.

Evaluation

In the main test, the total CPU usage of the node has been decreased by ~13%. Regarding the latency of the services, it shows an increase of around 18%, which does not exceed the model's known performance threshold. This optimization example is therefore stable. The four additional tests made in this environment show the resource management system's ability to be aware of the bad optimization step suggested and solve the conflict in the next iteration. Figure 5.1 shows the total CPU usage in node *worker-1* for the consumer services *a*, *b*, *c*. At 17:05 the quota specified in the agent was applied and the resource allocation was changed accordingly.

The following test cases are summarized and describe different situations for the three services.

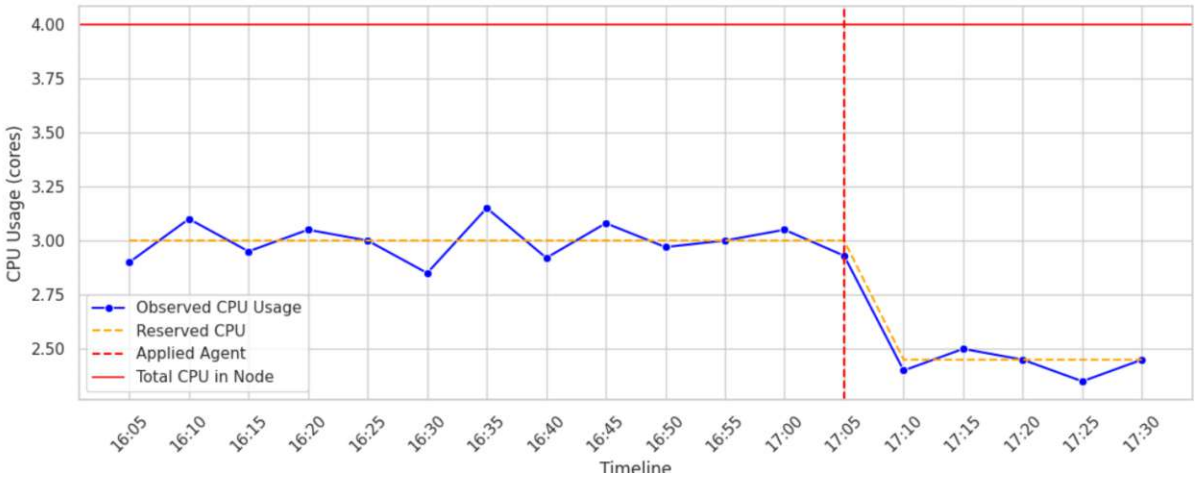

Figure 5.1: Changes in CPU usage in node *worker-1* after applying the resource quota specified by the agent.

5.3.2 Benchmark: Optimization of Overprovisioned Resources

In this benchmark, we intend to reduce the reserved memory in *worker-1* as well as the memory usage. Table 5.11 shows the high memory allocation of 81% on *worker-1*. The allocation is the sum of the memory requests of three services, which is 6480 MB (2840 + 2840 + 800), equivalent to the 6036.5 MiB specified in the configurations of the consumer services. The actual memory usage of 34% of the three services on the node is significantly lower than the allocated memory, which may imply overprovisioning. The services from the other worker nodes have remained with the identical, low-consumption configuration.

1. *Initial state:* Three services, one having a significantly lower memory usage.

   The following table shows the state of the worker nodes:

Node	# of Pods	Total CPU Usage (%)	Total Reserved CPU (%)	Total Memory Usage (%)	Total Reserved Memory (%)
worker-1	3	75	75	34	81
worker-2	2	50	50	0.57	37
worker-3	2	50	50	0.57	37
worker-4	2	50	50	0.57	37

Table 5.11: Overview of the node resources in the initial state.

2. *Agent Creation:* node-wide, memory quota of 30%.

3. *Optimization:* As can be seen in table 5.12, the DRL System optimized *consumer-a* and *consumer-b*, however, *consumer-c* was migrated to another node because the memory reduction implied having less memory available than required for the container to be able to run. The applied memory for consumer-c is inside brackets

Deployment Name	Replicas	Node	Init Mem	Applied Mem	Action
consumer-a	1	worker-1	1261.33	882.93	optimize
consumer-b	1	worker-1	1245.5	871.85	optimize
consumer-c	1	worker-1	197.54	(197.54)	migrate

Table 5.12: Deployment configuration overview with applied resources and actions.

because the algorithm does not apply any consumption changes if the DRL System suggests migration.

After applying the resource quota to the whole node, the following statistics can be seen:

Node	# of Pods	Total Usage (%)	Total Reserved CPU (%)	Total Memory Usage (%)	Total Reserved Memory (%)
worker-1	2	50	50	21.3	21.3
worker-2	2	50	50	0.57	37
worker-3	3	75	75	0.86	56
worker-4	2	50	50	0.57	37

Table 5.13: Overview of the resource usage and reserve decrease after applying the resource quota.

Worker-1 had a memory usage reduction of ~38%, from the initial 34% total usage to 21.3%. The actual memory usage reduction was higher because consumer-c has been migrated to worker-3. This can be observed in Table 5.13.

4. *Evaluation:* This benchmark has been run more times on similar setups and there have been cases, three in 20 tests, where the service with lower memory would be optimized instead of migrated, leaving it with less than 150 MiB RAM, which would lead to an OOMKill. Six in 20 cases included degraded performance of services *A* and *B*. However, the DRL System received bad feedback in these cases and corrected itself for future predictions.

   Figure 5.2 displays the total memory usage in node *worker-1* and how it is distributed across the three target services *A*, *B* and *C*. The changes in memory usage triggered by the agent can be seen starting at 17:55. the memory usage reduction of ~38% on the *worker-1* node is very important. In this case, reserved memory allocation was reduced from 81% to 21.3%, as displayed by the dotted orange line. The total reserved memory was reduced enough to contain the actual resource consumption used on this node. For this threshold to be respected, the memory limit in the Kubernetes deployment was set equal to the memory reserve. The bottom line represents the CPU usage of consumer *C*. The line stops at 17:50 where the agent has been applied because the service was moved to another node and

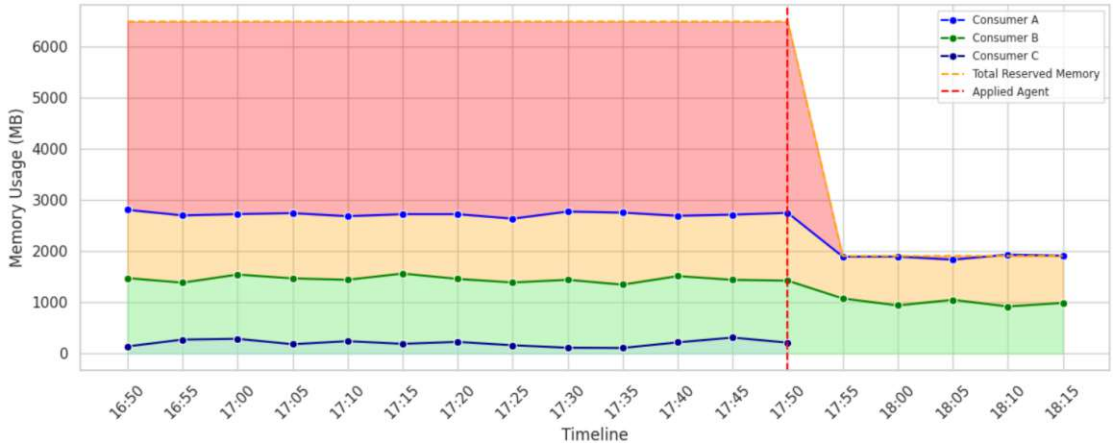

Figure 5.2: Sum of memory usage of node *worker-1* split between the three target services.

Prometheus stopped registering its usage data on *worker-1*. Overall, this procedure freed ~74% of the allocated memory, while respecting the configured performance thresholds of 20% allowed performance degradation. In case of a significantly higher latency, the watcher would have noticed the increase and reversed the optimization decision.

5.3.3 Benchmark: Replica Awareness and Handling

The main goal of this benchmark is to check the performance of the framework's decision-making when scaling services upon applying a high quota reduction.

1. *Initial state:* Two services, one having three replicas on the same node.
2. *Agent Creation:* node-wide, memory quota of 0.7%.

Deployment Name	Replicas	Node	Init Mem	Applied Mem	Action
consumer-a	3	worker-1	813.33	569,32	scale down
consumer-b	1	worker-1	253.24	177,27	optimize

Table 5.14: Deployment configuration overview with applied resources and suggested optimization actions.

3. *Optimization:* As can be seen in table 5.12, the DRL System first obtained the total memory consumption of all three replicas of *consumer-a* in the workload and reduced the usage not by vertically scaling the replicas, but by horizontally scaling down.

Additional Tests for Different Scaling Approaches

The following table describes further benchmarks for testing the optimization through replica scaling. These tests are focused only on single services running on a node with more replicas.

Test #	Replicas	Total Init CPU (cores)	CPU Quota (%)	Total Init Mem (MiB)	Mem Quota (%)	Action
1	4	3.96	0.75	648.71	100	scale down
2	4	3.96	0.82	648.71	100	optimize
3	2	1.00	1.0	393.73	55	scale down
4	2	1.85	1.0	820.50	40	scale down

Table 5.15: Further tests made to benchmark optimization through scaling.

As can be seen in Table 5.15, the resource management system correctly scaled down the service in test 1# upon receiving the 25% CPU reduction request. This shows that the model learned the correlation between the total resource usage and the number of replicas included in the service. With four cores allocated in four replicas, the model suggests scaling down the service by one since the optimization implies having only three cores.

Test 2# also shows the same service but with a higher CPU quota. In this case, the model suggests optimizing the deployment and not downscaling it.

Test 3# depicts a service with two replicas that requires its memory to be reduced by 55%. In this case, the model decides to scale down. Optimization may also have worked; however, it could have impacted the replicas' performance.

Evaluation of Optimization through Scaling

As seen in the main example and the additional tests, the model can successfully correlate the total resources available and the number of replicas included in the service. There are, however, arguable optimization steps in the DRL system's logic where it may not be sure in case of a significant resource reduction higher than the initial usage of only one replica of a service. In such cases, the DRL model may choose to scale down or evict the replicas altogether to another node. The latter would imply a behaviour similar to draining the node.

5.4 Overall Evaluation

This framework was implemented in this system to automatically optimize edge infrastructures with computing continuum workloads and solve optimization conflicts with the following key points in mind:

1. Maintaining the workflow of the services when optimizing to promote the computing continuum.

2. Designing the framework as simply as possible to also provide a good performance.
3. Providing an accurate DRL solution to optimize services and solve conflicts in case they occur.
4. Offering a quick resource optimization solution for services that use significantly fewer resources than initially allocated.
5. Providing a simple interface for the administrator to create, edit, and remove agents and thus simplify the process of triggering conflicts and optimizations.

The implemented resource management system respects this evaluation's first main point. With the innate feature included in Kubernetes, the *"RollingUpdate"* deployment strategy, the pods are never shut down until the next ones are successfully running. This implies that the workloads can run continuously until the next optimized service version is stable.

As mentioned in the second point in the previous paragraph, the system's performance is of high importance in this research. The time complexity of the watcher, which views and applies the existing agents to the targeted deployments depending on the various scopes, is linear. The following formula describes the time complexity of the algorithm:

$$O(A \times D),$$

where A describes the agents and D the deployments. Of course, this algorithm would scale polynomially with increasing numbers of agents and deployments; however, this would not be realistic in an edge infrastructure. With this system, it is even encouraged to have as few agents as possible to prevent overlapping configurations that may lead to conflicts and bad service performance.

The third goal of the resource management system is the performance of the DRL system in optimizing and solving conflicts that are triggered in optimization. This category depends solely on the following two factors: the DRL System's performance and the Watcher's algorithm.

The benchmarks show that the DRL System has the same performance when implemented in the resource management system, as well as when being tested alone, separately, after training. Bad predictions in optimization are normal because, as mentioned in the previous sections, the DRL model is trained with average thresholds of average services. Services may always exist that operate under different thresholds, which is why reinforcement learning was added to the system. As seen in the tests, the DRL system also performs well when receiving feedback and re-trying predictions on previous deployments because of its specialized models trained for each deployment that have different thresholds. It is also important to mention that the reinforcement system also scales with more deployments. However, it is important to keep in mind that the specialized model only requires around 500 kB.

The fourth goal of the framework was also respected and showed significant results in the benchmarks. The optimization feature of the DRL System is a very important aspect

because it optimizes the resource usage and allocation of services, thus significantly reducing the costs of having these resources allocated in the first place. Overprovisioning is an important issue that may occur both at the creation of a workflow and during the lifetime of the service if the demand for resources changes over time. Such changes can be detected by the framework and mediated with the help of the agents, the watcher, and the DRL system.

CHAPTER 6

Conclusion

6.1 Limitations

One of the biggest limitations of this research is the synchronization of the various metrics acquired by Prometheus. The scraping frequency can indeed be configured to happen more often, but in the created system, fetching the latest information regarding a pod's performance after an optimization step may take up to three minutes, which for some use-cases might not be optimal.

Some lightweight testing has been done on a virtual minikube cluster. However, these tries were limited by their simplicity since it is not yet supported to configure the number of cores per node, which in turn left the nodes sharing the same four cores from the computer on which the cluster has been set up. On the other hand, Minikube proved useful for testing the prototypes of the components more quickly before having them deployed into the main cluster.

The first prototype of the resource management system was set up on a local K3S cluster consisting of a virtual machine, an NVIDIA Jetson, and two Raspberry Pi. The problem in this environment was that the Raspberry Pi was too slow and would not have been a good environment for benchmarking the resource management system. The main reason for this was that the Raspberry Pi had slower cores and processing power, meaning that a deployment that may perform great on the NVIDIA Jetson node would perform significantly worse on a Raspberry Pi. In this environment, all services would end up on the NVIDIA Jetson if an optimization test was made.

Regarding scalability for bigger, non-edge infrastructures with more than 100 nodes, a node-wide agent per node would also work efficiently if more watchers are distributed across more nodes to handle more API calls in parallel. However, in this case, an agent manager with a higher scope should be created, which could apply resource quotas to groups of nodes at the same time.

6.2 Future Work

An important direction of this research may be having only a single DRL model that would re-train itself and be able to distinguish between the deployments. Using an embedding layer for each deployment was tested in this research. However, when applying reinforcement learning to the neural network tested with this configuration, the learned optimization step would also be predicted for other deployments with similar metrics. This, in turn, would negatively alter the decision weights of the model. An approach for this case would be to generate more training data that would describe a single deployment with its thresholds. By having such train sets prepared for more deployments, the DRL model would be able to distinguish between the deployments successfully with the help of the embedding layer, thus no longer requiring splitting the models for specialized reinforcement learning. However, this method would require a lot of supervision and fine-tuning to profile each deployment that needs to be managed. This would be a more specific framework, while the system we developed in this research was designed to be more centralized and not require additional interactions and supervision for each special deployment.

The security perspective must be analyzed before deploying the framework in production. In the implementation used in this research, privileged service accounts were created for the relevant components. The controller only requires permissions to create, edit and delete the agent CRDs, whereas the watcher needs a cluster role to edit deployments across namespaces. Therefore, future research needs to observe network aspects [DMDM23, HMM+24] and the watcher's security more closely, as well as explore various approaches (e.g., [SMD22]).

Additionally, having the whole resource management system dynamically adaptable for every Kubernetes-based cluster and having it defined within a Helm chart would be an important step in the system's further development.

6.3 Conclusion

This research began with a high-level description of the present situation where conflicts occurred between resource management agents that acted independently in edge environments and then set the goal of the thesis with the help of three main research questions.

To answer the research questions, a framework was required upon which the DRL Model would be implemented. This framework was a thorough blueprint for building a well-performing simulation environment on which resource management agents could quickly be created and edited. The resource quotas mentioned in their specifications can easily be applied to the Kubernetes cluster with the help of the Watcher and DRL system components.

Regarding the first research question, the main DLR model proved to be more of an advisor for the watcher. It predicts the best optimization step that may be used for a

deployment, regardless of whether it is already successfully running or in a conflicting state. The watcher is the component that, in synergy with the DRL model, can mediate resource management conflicts. To achieve this desired performance, the DRL model had to be trained using thousands of synthesized datasets from many deployments to teach the model the standard CPU and memory thresholds of containers. The latency of the problem-solving is not an issue for this framework because most of the conflicts that may be triggered in this environment are covered by the watcher and the DRL system. In case of agent configuration conflicts, the administrator is notified immediately in the dashboard.

To answer the second research question, a simple deep learning model would have been very problematic in this framework because it would not improve itself based on its mistakes. The reinforcement learning feature is the changing factor in the framework since the DRL model can adapt to services with different thresholds in case they have different resource usage patterns. For this, the DRL System was created, which can effectively mediate conflicts triggered by an agent's optimization task and suggest optimization steps for all the services the agent targets. The watcher oversees the performance of the DRL system. After the DRL system optimizes a deployment, the watcher observes how the deployment performs after the optimization process. Suppose the deployment performance degrades and reaches a specific threshold where the service's SLOs are breached. In that case, the watcher sends feedback with a negative rating to the DRL system, which in turn trains a clone of the meta DRL model to be specialized for this deployment only, in case future optimization is required.

This, in turn, answers the third research question. The main metric used for measuring the impact of the decisions taken by the DRL system is the latency metric, a custom metric that is dynamically exposed to the Prometheus service so that the resource management system can process it. The DRL model will always make a decision, and the watcher will oversee it and send feedback if needed to improve the model.

In conclusion, this research presents a new approach to resolving conflicts in resource management in edge computing environments. The suggested framework, which consists of the controller, watcher, and the DRL system, not only solves conflicts in such environments but also optimizes the service resource allocation by being aware of the resource consumption and performance of the services. Hand in hand with this feature is the possibility of simulating such conflicts in the sandbox functionality offered by the framework. Creating agents with specific resource management tasks and overseeing their impact on the cluster with the help of the dashboard is a powerful tool for distributing agents across wider infrastructures. As a main support to the optimization tasks performed within this framework, the DRL system makes sure to suggest the best optimization steps for the services as well as solve conflicts, in case the SLOs of the deployments are no longer respected. The reinforcement learning feature makes the DRL system highly adaptable to workload changes within the cluster, making it react to changes in resource requirements when they are made.

Source Code Repository

The full implementation of the Deep Reinforcement Learning Resource Management System (DRL-RMS) Framework developed as part of this thesis is available in the *DRL-RMS-Framework* repository[PV25] on GitHub:

`https://github.com/vladfreeze/DRL-RMS-Framework`

This repository contains the code used to set up the framework and documentation to support reproducibility and further research.

List of Figures

List of Tables

List of Algorithms

Bibliography

[Ku24] Kubernetes. *Kubernetes Documentation*. The Cloud Native Computing Foundation, 2024. Accessed: 2024-10-29.

[AABAH22] Mahfoudh Saeed Al-Asaly, Mohamed A Bencherif, Ahmed Alsanad, and Mohammad Mehedi Hassan. A deep learning-based resource usage prediction model for resource provisioning in an autonomic cloud computing environment. *Neural Computing and Applications*, 34(13):10211–10228, 2022.

[AADJA16] Mahmoud Al-Ayyoub, Mustafa Daraghmeh, Yaser Jararweh, and Qutaibah Althebyan. Towards improving resource management in cloud systems using a multi-agent framework. *International Journal of Cloud Computing*, 5(1-2):112–133, 2016.

[AMD20] Cosmin Avasalcai, Ilir Murturi, and Schahram Dustdar. Edge and fog: A survey, use cases, and future challenges. *Fog Computing: Theory and Practice*, pages 43–65, 2020.

[ASY20] Ali Asghari, Mohammad Karim Sohrabi, and Farzin Yaghmaee. A cloud resource management framework for multiple online scientific workflows using cooperative reinforcement learning agents. *Computer Networks*, 179:107340, 2020.

[BLYZ21] Jing Bi, Shuang Li, Haitao Yuan, and MengChu Zhou. Integrated deep learning method for workload and resource prediction in cloud systems. *Neurocomputing*, 424:35–48, 2021.

[BMDD23] Cem Bicer, Ilir Murturi, Praveen Kumar Donta, and Schahram Dustdar. Blockchain-based zero trust on the edge. In *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1006–1013. IEEE, 2023.

[Cha23] Shireen Chand. Choosing between cross entropy and sparse cross entropy — the only guide you need!, July 2023. Accessed: 2025-04-15.

[CK13] Taha Chaabouni and Maher Khemakhem. Resource management based on agent technology in cloud computing. In *2013 Taibah University International Conference on Advances in Information Technology for the Holy Quran and Its Sciences*, pages 372–375. IEEE, 2013.

[CNC25] CNCF. Distribution registry. `https://distribution.github.io/distribution/`, 2025. Accessed: 2025-04-14.

[CPDM+23] Victor Casamayor Pujol, Praveen Kumar Donta, Andrea Morichetta, Ilir Murturi, and Schahram Dustdar. Distributed computing continuum systems–opportunities and research challenges. In *Service-Oriented Computing–ICSOC 2022 Workshops: ASOCA, AI-PA, FMCIoT, WESOACS 2022, Sevilla, Spain, November 29–December 2, 2022 Proceedings*, pages 405–407. Springer, 2023.

[CR+11] Rodrigo N. Calheiros, Rajkumar Buyya Ranjan, et al. Cloudsim: A framework for modeling and simulation of cloud computing infrastructures and services, 2011. Accessed: YYYY-MM-DD.

[CYY+17] Chia-Chen Chang, Shun-Ren Yang, En-Hau Yeh, Phone Lin, and Jeu-Yih Jeng. A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[DM20] Schahram Dustdar and Ilir Murturi. Towards distributed edge-based systems. In *2020 IEEE Second International Conference on Cognitive Machine Intelligence (CogMI)*, pages 1–9. IEEE, 2020.

[DM21] Schahram Dustdar and Ilir Murturi. Towards iot processes on the edge. In *Next-Gen Digital Services. A Retrospective and Roadmap for Service Computing of the Future: Essays Dedicated to Michael Papazoglou on the Occasion of His 65th Birthday and His Retirement*, pages 167–178. Springer, 2021.

[DMDM23] Praveen Kumar Donta, Edmundo Monteiro, Chinmaya Kumar Dehury, and Ilir Murturi. Learning-driven ubiquitous mobile edge computing: Network management challenges for future generation internet of things. *International Journal of Network Management*, 33(5), 2023.

[DPD22] Schahram Dustdar, Victor Casamayor Pujol, and Praveen Kumar Donta. On distributed computing continuum systems. *IEEE Transactions on Knowledge and Data Engineering*, 35(4):4092–4105, 2022.

[GLU20] Pengsheng Guo, Chen-Yu Lee, and Daniel Ulbricht. Learning to branch for multi-task learning. In *International conference on machine learning*, pages 3854–3863. PMLR, 2020.

[gra] Grafana documentation - latest version. Online documentation. Accessed: 2025-03-23.

[HMM+24] Abhishek Hazra, Andrea Morichetta, Ilir Murturi, Lauri Lovén, Chinmaya Kumar Dehury, Víctor Casamayor Pujol, Praveen Kumar Donta, and Schahram Dustdar. Distributed ai in zero-touch provisioning for edge networks: Challenges and research directions. *Computer*, 57(3):69–78, 2024.

[Inf24] InfluxData. *InfluxDB Documentation*, 2024. Accessed: 2024-10-30.

[jup] Jupyter documentation - latest version. Online documentation. Accessed: 2025-03-23.

[K3s] K3s Project Authors. *K3s Documentation*. The Linux Foundation. Accessed: 2024-12-04.

[KMD24] Daniel Kaltenböck, Ilir Murturi, and Schahram Dustdar. A zero trust single sign-on framework with attribute-based access control. In *2024 26th International Conference on Business Informatics (CBI)*, pages 149–157. IEEE, 2024.

[lea] Kubernetes: Deploying a docker registry on kubernetes. Leaseweb Knowledge Base. Accessed: 2025-03-23.

[LLX+17] Ning Liu, Zhe Li, Jielong Xu, Zhiyuan Xu, Sheng Lin, Qinru Qiu, Jian Tang, and Yanzhi Wang. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*, pages 372–382. IEEE, 2017.

[LMR+17] John Lawrence, Jonas Malmsten, Andrey Rybka, Daniel A Sabol, and Ken Triplin. Comparing tensorflow deep learning performance using cpus, gpus, local pcs and cloud. 2017.

[MATD19] Ilir Murturi, Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. Edge-to-edge resource discovery using metadata replication. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–6. IEEE, 2019.

[MD21] Ilir Murturi and Schahram Dustdar. A decentralized approach for resource discovery using metadata replication in edge networks. *IEEE Transactions on Services Computing*, 2021.

[MD22] Ilir Murturi and Schahram Dustdar. Decent: A decentralized configurator for controlling elasticity in dynamic edge networks. *ACM Transactions on Internet Technology (TOIT)*, 2022.

[MDP+23] Ilir Murturi, Praveen Kumar Donta, Victor Casamayor Pujol, Andrea Morichetta, and Schahram Dustdar. Learning-driven zero trust in distributed computing continuum systems. In *2023 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pages 0044–0049. IEEE, 2023.

[Met] MetalLB Project. *MetalLB Documentation*. Accessed: 2024-12-04.

[MNPC20] Octavian Mart, Catalin Negru, Florin Pop, and Aniello Castiglione. Observability in kubernetes cluster: Automatic anomalies detection using prometheus. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 565–570. IEEE, 2020.

[NCN+22] Ovidiu-Constantin Novac, Mihai Cristian Chirodea, Cornelia Mihaela Novac, Nicu Bizon, Mihai Oproescu, Ovidiu Petru Stan, and Cornelia Emilia Gordan. Analysis of the application efficiency of tensorflow and pytorch in convolutional neural network. *Sensors*, 22(22):8872, 2022.

[NSSN21] Navid Naderializadeh, Jaroslaw J Sydir, Meryem Simsek, and Hosein Nikopour. Resource management in wireless networks via multi-agent deep reinforcement learning. *IEEE Transactions on Wireless Communications*, 20(6):3507–3523, 2021.

[Ope24] OpenAI. Openai api pricing. `https://openai.com/api/pricing`, 2024. Accessed: 2024-12-09.

[PK17] Sungheon Park and Nojun Kwak. Analysis on the dropout effect in convolutional neural networks. In *Computer Vision–ACCV 2016: 13th Asian Conference on Computer Vision, Taipei, Taiwan, November 20-24, 2016, Revised Selected Papers, Part II 13*, pages 189–204. Springer, 2017.

[PNW20] Bo Pang, Erik Nijkamp, and Ying Nian Wu. Deep learning with tensorflow: A review. *Journal of Educational and Behavioral Statistics*, 45(2):227–248, 2020.

[PV25] Vlad Popescu-Vifor. DRL-RMS-Framework: Deep Reinforcement Learning for Resource Management System Framework. `https://github.com/vladfreeze/DRL-RMS-Framework`, 2025. Accessed: 2025-05-14.

[rke] Rke2 - the secure kubernetes engine. Online documentation. Accessed: 2025-03-23.

[SB19] Nitin Sukhija and Elizabeth Bautista. Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus. In *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 257–262. IEEE, 2019.

[SMD22] Boris Sedlak, Ilir Murturi, and Schahram Dustdar. Specification and operation of privacy models for data streams on the edge. In *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*, pages 78–82. IEEE, 2022.

[SOE18] 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.

[SUS24] SUSE. Harvester: Open Source HCI Solution, 2024. Accessed: 2024-11-04.

[The25] The Kubernetes Authors. Minikube documentation, 2025. Accessed: 2025-04-14.

[TMD19] Christos Tsigkanos, Ilir Murturi, and Schahram Dustdar. Dependable resource coordination on the edge at runtime. *Proceedings of the IEEE*, 107(8):1520–1536, 2019.

[WB13] Yi Wei and M Brian Blake. Adaptive service workflow configuration and agent-based virtual resource management in the cloud. In *2013 IEEE International Conference on Cloud Engineering (IC2E)*, pages 279–284. IEEE, 2013.

[ZGP+19] Deze Zeng, Lin Gu, Shengli Pan, Jingjing Cai, and Song Guo. Resource management at the network edge: A deep reinforcement learning approach. *IEEE Network*, 33(3):26–33, 2019.

[ZTB+18] Iffat Zafar, Giounona Tzanidou, Richard Burton, Nimesh Patel, and Leonardo Araujo. *Hands-on convolutional neural networks with TensorFlow: Solve computer vision problems with modeling in TensorFlow and Python*. Packt Publishing Ltd, 2018.