Transferência de Aprendizado por Reforço para Elasticidade de Serviço de Nuvem

Ian Resende da Cunha

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2024

Ian Resende da Cunha

Transferência de Aprendizado por Reforço para Elasticidade de Serviço de Nuvem

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Rafael Pasquini
Coorientador: Matías Richart

Uberlândia
2024

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

C972 2024

Cunha, Ian Resende da, 1997-
Transfer of Deep Reinforcement Learning for Cloud Service's Elasticity [recurso eletrônico] / Ian Resende da Cunha. - 2024.

Orientador: Rafael Pasquini.
Coorientador: Matías Richart.
Dissertação (Mestrado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.di.2024.173
Inclui bibliografia.
Inclui ilustrações.

1. Computação. I. Pasquini, Rafael,1983-, (Orient.). II. Richart, Matías,1987-, (Coorient.). III. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. IV. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.

Agradecimentos

Agradeço ao meu orientador, Professor Dr. Rafael Pasquini, pela valiosa orientação, confiança e compreensão ao longo de todo o trabalho.

Ao meu coorientador, Professor Dr. Matías Richart, e ao Professor Dr. Javier Baliosian, que colaboraram fortemente para o desenvolvimento deste trabalho, sempre disponíveis para discussões e sugestões.

À minha noiva Isadora, dedico especial reconhecimento pela sua paciência incansável e apoio incondicional durante esta jornada. Por restaurar minhas forças e motivação nos momentos mais difíceis, por compreender minha ausência em inúmeras ocasiões e por ser a melhor companheira que eu poderia pedir. Este trabalho também é seu.

Aos meus pais, Márcio e Edilaine, meu agradecimento por todo o apoio, incentivo e compreensão em todas as situações. Seu sacrifício e dedicação em oferecer-me a melhor educação possível, tanto pessoal quanto acadêmica, são inestimáveis.

Aos meus familiares que me apoiam e desejam o melhor para mim.

À Noussec, empresa onde trabalho, por aceitar e apoiar constantemente a realização deste trabalho.

E, por fim, àqueles professores da Faculdade de Computação que fizeram mais do que seu trabalho, transmitindo e despertando em mim o fascínio pela ciência e o desejo de colaborar nessa jornada de ensino e principalmente constante aprendizado.

"You have power over your mind, not outside events.
Realize this, and you will find strength."
(Marcus Aurelius)

Resumo

O gerenciamento de recursos em ambientes de computação de nuvem é um desafio crítico, no qual um mecanismo de orquestração busca garantir a utilização otimizada de recursos, mantendo a qualidade do serviço, evitando desperdícios e reduzindo custos. Uma abordagem promissora para automatizar essa tarefa envolve o uso de técnicas de aprendizado de máquina. No entanto, essa abordagem também enfrenta desafios no treinamento online em ambientes reais, relacionados à complexidade dos sistemas, longas durações de treinamento, restrições de segurança e rigidez do sistema. Esta dissertação tem como objetivo aprimorar e viabilizar o processo de treinamento de Aprendizado por Reforço para tarefas relacionadas à orquestração de recursos de serviços de nuvem, utilizando a Transferência de Aprendizado (*Transfer Learning* - TL). Um ambiente fonte foi construído, composto por um modelo de simulação do serviço, e o conhecimento adquirido em treinamento em simulação foi transferido para aprimorar o novo treinamento no ambiente do mundo real. Uma análise comparativa entre TL e métodos de treinamento tradicionais apresenta resultados positivos, incluindo uma redução substancial no tempo necessário para alcançar um desempenho razoável, melhorias de até 40% no desempenho inicial dos agentes e um aprimoramento de até 30% no desempenho geral durante as fases de treinamento e teste. Por fim, foi demonstrado que um agente treinado em simulação pode ser reutilizado diretamente no ambiente real sem treinamento adicional, produzindo resultados satisfatórios e consistentes.

Palavras-chave: Transferência de Aprendizado. Aprendizado por Reforço Profundo. Aprendizado de Máquina. Orquestração. Gerenciamento de Recursos. Elasticidade. Serviço de Nuvem. Banco de dados Cassandra.

Transfer of Deep Reinforcement Learning for Cloud Service's Elasticity

Ian Resende da Cunha

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2024

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da Computação
Av. João Naves de Ávila, 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Dissertação de Mestrado, 9/2024, PPGCO				
Data:	29 de fevereiro de 2024	Hora de início:	09:00	Hora de encerramento:	12:30
Matrícula do Discente:	12012CCP003				
Nome do Discente:	Ian Resende da Cunha				
Título do Trabalho:	Transfer of Deep Reinforcement Learning for Cloud Service's Elasticity				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Sistemas de Computação				
Projeto de Pesquisa de vinculação:	SFI2 - Slicing Future Internet Infrastructure				

Reuniu-se por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Matías Mario Richart Gutiérrez- University of the Republic in Uruguay (Coorientador), Rodrigo Sanches Miani - FACOM/UFU, Rodolfo da Silva Villaca - PPGI/UFES e Rafael Pasquini - FACOM/UFU, orientador do candidato.

Os examinadores participaram desde as seguintes localidades: Matías Mario Richart Gutiérrez - Barcelona/Espanha e Rodolfo da Silva Villaca- Vitória/ES . Os outros membros da banca e o aluno participaram da cidade de Uberlândia.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Rafael Pasquini, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir a senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir ao candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Mestre.

Ressalta-se que o Coorientador Matías Mario Richart Gutiérrez, por ser estrangeiro, residente em outro país e não possuir CPF registrado no Brasil não assinará a ata de defesa.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Rafael Pasquini**, **Professor(a) do Magistério Superior**, em 04/03/2024, às 09:21, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Rodrigo Sanches Miani**, **Professor(a) do Magistério Superior**, em 04/03/2024, às 10:57, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **RODOLFO DA SILVA VILLACA**, **Usuário Externo**, em 04/03/2024, às 15:59, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5195411** e o código CRC **46A98E1C**.

Abstract

Resource management in cloud computing environments is a critical challenge, where an orchestration mechanism seeks to ensure optimized resource utilization while maintaining service quality, preventing waste, and reducing costs. A promising approach to automating this task involves employing machine learning techniques. However, this approach also faces challenges in real-world online training, related to system complexity, extended training durations, safety restrictions, and system rigidity. This dissertation aims to enhance and streamline the Deep Reinforcement Learning training process for tasks related to the orchestration of cloud service resources by employing the Transfer Learning (TL) technique. A source environment was built, comprising a simulation of the target service, and knowledge acquired through simulation training was transferred to enhance training in the real-world service environment. Comparative analysis between TL-based and standard training reveals positive outcomes, including a substantial reduction in time required to achieve reasonable performance, improvements of up to 40% in the initial performance of agents, and up to a 30% enhancement in overall performance during training and testing phases. Finally, it was demonstrated that an agent trained in simulation could be deployed directly into the real environment without additional training, yielding satisfactory and consistent outcomes.

Keywords: Transfer Learning. Deep Reinforcement Learning. Machine Learning. Orchestration. Resource Management. Elasticity. Cloud Service. Cassandra Database.

List of Figures

List of Tables

Acronyms list

AI Artificial Intelligence

DRL Deep Reinforcement Learning

DQN Deep Q-Networks

DNN Deep Neural Networks

IMA Infrastructure & Monitoring Abstraction

ML Machine Learning

NFV Network Function Virtualization

QoS Quality of Service

REM Real Environment Module

RL Reinforcement Learning

RO Resource Orchestrator

RNN Recurrent Neural Networks

SEM Simulation Environment Module

SDN Software-Defined Networking

SRO Slice Resource Orchestrator

TL Transfer Learning

VM Virtual Machines

Contents

I hereby certify that I have obtained all legal permissions from the owner(s) of each third-party copyrighted matter included in my thesis, and that their permissions allow availability such as being deposited in public digital libraries.

Ian Resende da Cunha

Chapter 1

Introduction

The rise of cloud computing has revolutionized the way we store, manage, and access data and applications. Cloud computing provides a flexible and scalable way to provision computing resources on demand, allowing users to access advanced computing resources without investing in hardware or local infrastructure. As such, it has become a vital part of the daily operations of modern businesses and has been widely adopted across various sectors, types, and sizes of companies (EL-GAZZAR, 2014).

However, as the demand for cloud computing services continues to grow, with high demands for scalability, availability, and security, the challenges of efficiently managing cloud resources also increase. One of these challenges is the orchestration of cloud computing resources, which involves managing and allocating these resources efficiently and cost-effectively, aiming to cope with the dynamism of the demand, which is influenced by the variety of types of infrastructure, environments, technologies, tools, and software (JENNINGS; STADLER, 2015).

Therefore, this research focuses on the role of the Resource Orchestrator (RO) in the context of cloud computing and networking infrastructures, the RO is responsible for managing and scaling the allocation of resources. The management of cloud computing resources could be performed manually, with human-operated adjustments, by monitoring the structure's state and adjusting the allocated resources according to the varying demand.

Another approach that provides a primary level of automation to resource management would be with human-generated heuristics policy, defining and configuring rules that could guide the resource orchestration process (BELOGLAZOV; ABAWAJY; BUYYA, 2012). These heuristics establish, for example, guidelines or thresholds that command when to perform automatic changes in the level of allocated resources. For instance, when the level of demand received by a service exceeds the upper-bound value, the platform understands that it must automatically allocate more resources to the service. Likewise, when the received demand is lower than the lower-bound value, the platform should release unnecessary resources.

Furthermore, promising approaches employ Artificial Intelligence (AI) and Machine Learning (ML) techniques in the pursuit of optimized and efficient resource management and orchestration (MAO et al., 2016; STADLER; PASQUINI; FODOR, 2017; LI et al., 2018). Thus, solutions are researched seeking to mitigate related issues, such as service degradation during peak demand periods, and to provide greater cost savings by using fewer unnecessary resources.

This work focuses on a ML approach, specifically employing a Deep Reinforcement Learning (DRL) Algorithm. Reinforcement Learning (RL), an interdisciplinary area of ML and AI, enables learning from trial-and-error experiences in real-time environments. By continuously optimizing their decision-making based on rewards and penalties, RL presents a promising approach to address the outlined challenge.

RL presents continuous and online learning, constantly updating the ML model, its functions, and policies during utilization. As new states and metrics patterns are perceived, the ML model is adjusted, adapting to the environmental changes. In this way, we expect that proposing the utilization of an RL algorithm will enable real-time updates of the ML model, potentially enhancing its versatility, reliability, and resilience.

In recent years, RL has shown remarkable success in several domains, from playing games to autonomous driving systems (OPENAI et al., 2019; KIRAN et al., 2022). However, based on the experience we obtained during this research, corroborated by related research (WANG et al., 2017; QIU et al., 2023; ZHU et al., 2023), we verified that training RL models directly in real-world practical and critical infrastructures presents several additional challenges and limitations because of environmental complexity and operational characteristics of large-scale and dynamic services and environments, such as:

One critical challenge lies in the system's time to execute actions and delayed feedback. Actions do not occur immediately. For instance, it takes time to complete a service node addition or removal from a cluster. Additionally, the action's effects in the system, whether positive or negative, are not promptly observed, requiring a time delay. This time depends on factors such as stored data volume and bandwidth, potentially leading to prolonged training periods, which may take weeks to reach the desired performance level.

Additionally, security concerns impose relevant limitations to training, especially in critical systems that cannot tolerate downtime or service degradation. The trained agent must explore the diverse states of the environment, which can lead to failure and undesired states, especially at the beginning of the training, endangering the integrity of the service, environment, and data and even posing risks to human safety.

Therefore, investigations towards facilitating the training process in related contexts of management of cloud resources in real-world settings are necessary and relevant for the present scenario. Thus, it requires the pursuit of alternatives that could make the process of training models simpler, faster, and safer.

From this premise arises the main proposal of this research, where the primary objective is to test and validate a technique that could improve the training process of RL agents over complex scenarios such as real cloud infrastructure environments, employing the so-called Transfer Learning (TL) technique (PAN; YANG, 2010). Seeking to improve and make feasible the training of tasks that would otherwise be risky, costly, or impractical in a real-world setting.

Essentially, transfer learning is a machine learning technique that involves the utilization of knowledge acquired in a source domain to somehow benefit the training and learning process of an agent in a new target task or domain (TAYLOR; STONE, 2009).

Therefore, within the context of this research, we investigate the interaction between a simulation service environment and a real-world service environment. The main objective is to reuse previous knowledge consolidated in a model trained over a simpler and controlled environment (simulation), by transferring it to the real environment. As a use case, we adopted a key-value store named Cassandra (Apache, 2016), given that:

1. Cassandra supports elasticity by automatically adjusting the key space when increasing or decreasing the cluster;

2. We found a simulation modeling of Cassandra (DIPIETRO; CASALE; SERAZZI, 2017) that we could extend to our context by implementing it in Simulink (MATHWORKS, 2020);

3. Our previous research experience using Cassandra deployments (CUNHA, 2019; MARQUES et al., 2019; REZENDE, 2020).

In summary, this research investigates the intersection of the two concepts: RL and TL. We propose a model training strategy that can mitigate some of the major challenges encountered in training such models in real-world, critical, and complex environments. Thus, experiments will not be restricted to simulated environments, but also, and mainly, we will investigate the previously mentioned effects on a real-world environment specifically built for our tests.

1.1 Research Goals and Challenges

The primary goal of this work is to demonstrate that, with a transfer learning technique, it is possible to enhance the RL training process in terms of duration and quality, within the context of resource orchestration in a real distributed cloud service.

Specific objectives that are integrated into the general objective of the research are:

- ❑ G1. Define and implement the necessary elements for the creation of a testbed that allows the validation of the objectives of this research;

- ❏ G2. Demonstrate the feasibility of conducting training RL models in both constructed simulation and real-world environments;
- ❏ G3. Demonstrate that the RL agent can orchestrate the elasticity of Cassandra, by coordinating the adjusting of the number of active nodes in the service cluster, balancing the cost/performance relationship;
- ❏ G4. Evaluate the transfer learning process by comparing the performance and duration of the standard training, without transfer, with the training enriched by the knowledge pre-established in simulation.
- ❏ G5. Demonstrate through experimentation the behavior of our research proposal, highlighting key findings on the topics of reinforcement learning and transfer learning.

1.2 Hypothesis

Given the problems and challenges foreseen in this research, we raise the following questions:

- ❏ Is it viable to integrate a simulation of Cassandra DB with an RL algorithm so that the environment is suitable for model training? In the same way, is it viable for a real environment?
- ❏ Is it possible to train tasks in both environments such that the RL training performance converges with satisfactory performance?
- ❏ Is it possible to automate the orchestration of a service resource in order to utilize the minimum necessary resources and meet the proposed Quality of Service (QoS) level in this specific context?
- ❏ Can transfer learning from a simulation environment provide benefits to the RL training duration and performance in the real-world environment?

Therefore, hypotheses are formulated to guide the development of the work.

H1. The transfer of knowledge acquired in a source simulation environment, enhances the real-world environment RL training process, in terms of duration or performance.

H2. The RL agent learns to optimize the cost/performance relationship of the infrastructure by dynamically balancing the number of active nodes in a Cassandra service cluster.

H3. An agent trained only in the source simulation environment, deployed into the real-world environment without additional training, provides prompt and reasonable performance.

1.3 Outline

This dissertation is organized as follows: Chapter 2 provides details of the background and related work. Chapter 3 describes the research proposal, the experimental workflow and the environment architecture. Chapter 4 provides details on the construction of the testing environment and defines the testing parameters and configurations, while Chapter 5 present the experiments and results obtained by the TL training. Finally, Chapter 6 presents our conclusions and future work.

Chapter 2

Fundamentals

In this chapter, we present and discuss the key concepts and definitions that underpin our research. Additionally, we review the related work, offering a concise description and analysis of the selected studies.

2.1 Cloud Resource Management

Analogous to on-premise computing and network infrastructures, cloud computing environments and their hosted services are supported by a diverse set of computational resources. These resources include memory, processing power, network bandwidth, and storage, as well as computing instances such as Virtual Machines (VM), containers, network slices, or service nodes. Cloud resource management involves the intelligent allocation and utilization of these resources, which is crucial for maintaining performance, optimizing costs, and ensuring scalability.

Cloud computing has introduced several new concepts and techniques that were previously unfeasible with traditional local computing infrastructures. One of these attributes is elasticity, which refers to the cloud's ability to dynamically allocate and deallocate resources based on current workload demand (AL-DHURAIBI et al., 2018). Elasticity can be categorized into:

- Horizontal Elasticity: Involves adding or removing complete instances (e.g., virtual machines or containers) to scale in or out. For instance, resources can be managed by increasing or decreasing the number of similar computational active nodes of a distributed service.

- Vertical Elasticity: Where adjustments are made to the resource capacity of existing instances, such as memory, processing power, storage, and network capacity, thus, scaling up or scaling down the allocated resources.

Elasticity facilitates effective cloud resource management, where one of the primary objectives is to balance cost and performance. Resource management is a significant challenge in the cloud domain, where is important for tenants to ensure that their allocated infrastructure and resource can maintain the quality of hosted systems during peak demands while minimizing resource usage to save on financial and energy costs.

Straightforwardly, this management could be done manually, with human-operated adjustments made according to the varying needs for resources. Alternatively, autoscaling is a practice aimed at automatically adjusting the number of computational resources allocated to an application based on real-time demand. One common approach that provides a primary level of automation to resource management is the human-generated heuristics policy or rule-based autoscaling (BELOGLAZOV; ABAWAJY; BUYYA, 2012). This involves defining and configuring rules that guide the resource orchestration process. These heuristics establish, for example, guidelines or thresholds indicating when to perform automatic changes in the level of allocated resources.

Furthermore, promising approaches employ AI and ML techniques in the pursuit of optimized and efficient resource management (MAO et al., 2016; STADLER; PASQUINI; FODOR, 2017; LI et al., 2018) known as AI-based autoscaling.

The challenge of managing cloud resources falls into the classic problem of automatic control or autonomic computing. Kephart e Chess (2003) abstracted this problem as a control loop consisting of Monitoring, Analysis, Planning, and Execution phases, the MAPE loop, developed to provide a structured approach for creating self-managing systems.

The four phases of the MAPE loop continuously repeat itself, ensuring that the system dynamically adapts to changes and maintains performance without human intervention (QU; CALHEIROS; BUYYA, 2018):

- ❏ Monitoring: Metrics on resource usage, performance, and other relevant parameters are continuously gathered to provide an up-to-date view of the system's state.
- ❏ Analysis: The collected data is analyzed to detect trends, patterns, and anomalies. The goal is to understand system behavior and identify any deviations from normal operations that may require intervention.
- ❏ Planning: Based on the analysis, this step involves creating a plan to address any detected issues or optimize system performance.
- ❏ Execution: The final step involves executing the planned actions. This may involve adjusting resource allocations, deploying additional resources, or making other operational changes to ensure the system effectively adapts to current demands.

AI and ML techniques are well-suited to the MAPE framework, where their attributes can assist or even fully manage the various phases of the loop. For instance, REZENDE

(2020) presents a resource orchestrator approach employing ML techniques to facilitate the development of the Analysis and Planning phases of the MAPE loop by predicting service quality metrics, detecting anomalies, and planning necessary changes to avoid service degradation or idle resources.

Certain types of ML algorithms, such as the RL algorithms, present the potential to be integrated into the entire MAPE loop for cloud resource management, as RL methods also consist of a training loop with similar stages :

- ❏ Monitor: The initial step in RL involves obtaining an observation of the environment state from relevant metrics that describe the current system state. Additionally, RL can optimize monitoring by learning which metrics are most critical for performance and focusing on those.

- ❏ Analyze: RL employs a reward or penalty system based on the state observations and a predefined reward function, in this way the algorithm learns from the historical set of information and previous experiences, continually enhancing the agent's policy and its capacity to analyze current data to identify anomalies and determine if any action is necessary.

- ❏ Plan: The RL algorithm learns optimal resource allocation policies through extensive interactions with the environment. Also, multiple objectives can be balanced, such as minimizing costs while maximizing performance, thus defining how actions should be taken.

- ❏ Execute: RL agents are embedded into the environment and interact with it by executing actions, essential for their trial-and-error learning process. They can automate the execution of plans by dynamically adjusting resource allocations, scaling services, and modifying configurations in real-time.

In this work, we will employ a RL algorithm in the context of cloud resource management, considering the MAPE loop and its phases for an autonomic computing system, pursuing the creation of a suitable environment for conducting the primary investigation into transfer learning in a real-world environment.

2.2 Machine Learning

Machine learning refers to the field of study in AI and computer science that focuses on the development of algorithms and models that enable the learning from vast data to make predictions or decisions without being explicitly programmed. Machine learning involves statistical techniques and algorithms that allow models or agents to improve their performance on a task through experience, by identifying patterns and relationships in

data. This scientific field encompasses different categories, including Supervised Learning, Unsupervised Learning, and Reinforcement Learning.

Supervised Learning algorithms are trained using labeled data, meaning each input or register has its known corresponding output. During the training phase, the algorithm receives a set of training inputs along with their respective outputs. The algorithm analyzes the relationship between input and output, generating a model or function capable of estimating the labels of an unknown input dataset. In the testing phase, the generated model is used to estimate the output values of the inputs in the unknown test dataset.

The Unsupervised Learning algorithm must learn from an unlabelled dataset, meaning the collected data has no information about the respective outcome, and the label is unknown for the training and testing. The algorithm explores the dataset, attempting to identify patterns and form groupings among the data. These algorithms are used to classification, segment text topics, recommend items, and identify outliers in the data.

While also utilizing unlabeled data, Reinforcement Learning diverges from other machine learning methods in that it does not rely on pre-defined data. Instead, it gathers information in real time by interacting with the environment through a trial-and-error learning process that employs a system of actions and rewards.

2.2.1 Reinforcement Learning

Reinforcement learning (KAELBLING; LITTMAN; MOORE, 1996) involves training agents to make decisions through interaction with an environment. In RL, an agent learns by taking actions within an environment and receiving feedback in the form of rewards or penalties. The primary objective is for the agent to develop a strategy, referred to as a policy, that maximizes cumulative rewards over time.

The RL process involves a continuous interaction between an agent and its environment. As depicted in Figure 1, the training process initiates with the agent being in a state St and executing an action At within the environment.

This action induces a transition to a new state $St + 1$, resulting in a reward $Rt + 1$. Consequently, the agent learns from each interaction with the environment, analyzing the feedback that may be positive, neutral, or negative in relation to the action taken. This iterative process forms the foundation of the agent's learning, gradually contributing to the refinement of its decision-making strategy.

Key aspects and components of RL include:

1. State: The current situation or configuration of the environment. The state represents what the agent perceives and is a determinant aspect of the algorithm. It defines the space of action, main variables, and operation of the test environment.

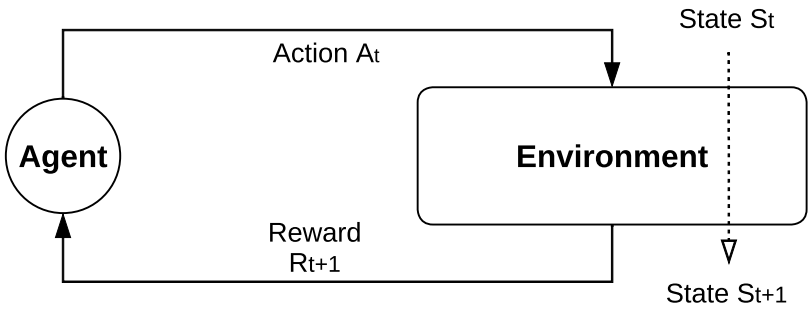

Figure 1 – Traditional reinforcement learning flow

2. Actions: The decisions or moves available to the agent. Actions define the possible operations the agent can execute within the training environment to interact with it.

3. Reward: A feedback value that the environment provides to the agent after it takes an action. The objective of the agent is to learn a policy that maximizes the cumulative reward over time.

4. Reward Function: The reward function is one of the most relevant aspects of effective training of RL models. The function must be able to describe and shape the desired behavior of the agent, positively rewarding those actions that lead to a desired result for the system and penalizing those that degrade the state of the environment.

Traditional RL demonstrated to be effective in fully observable environments with discrete state and action spaces tasks. However, when confronted with highly complex domains, especially those involving continuous and high-dimensional state spaces, it encounters difficulties (ARULKUMARAN et al., 2017; ZHU et al., 2023).

2.2.2 Deep Reinforcement Learning

In response to this limitation, an approach known as DRL has emerged (ARULKUMARAN et al., 2017). What distinguishes DRL is its integration of Deep Neural Networks (DNN) (SCHMIDHUBER, 2015) into the training process of RL agents, enabling them to navigate more challenging and complex domains with the ability to learn function approximators. DRL is a subject of research in diverse domains, from game playing (LAMPLE; CHAPLOT, 2017; OPENAI et al., 2019) to autonomous driving systems (KIRAN et al., 2022).

Despite the advancements in Deep Reinforcement Learning (DRL), its application to real-world domains presents challenges that necessitate further research and innovative solutions. Real-world domains often present unknown or partially observable environments. This necessitates extensive exploration of the system states, especially in the early stages

of training, until sufficient information is acquired to enable adequate exploitation of the knowledge and prevent suboptimal convergence.

This extensive exploration and information-gathering process in practical environments presents relevant limitations, such as sparse feedback, when the impact of an action is delayed or infrequent, extending the time needed to training instances. Additionally, high-dimensional state and action spaces further emphasize the vast number of interactions required to comprehend the environment, potentially making it an impractical activity within any reasonable timeframe. Finally, the nature of critical production environments prohibits exposure to the inherent risks of the initial exploratory training process. This process can lead the system to failure states, potentially causing system or service damage, unavailability, and safety concerns (ZHU et al., 2023; QIU et al., 2023).

Thus, the collection of experience through exploratory interactions in real-world environments becomes a challenge, not only in terms of time but also in terms of safety, particularly in domains where poor decisions/actions can lead to failure states in critical systems such as those related to health, finance, automated vehicles, and government.

Consequently, the pursuit of advancements and novel solutions for the reinforcement learning process, especially in the context of real-world environments, as mentioned above, remains relevant. One of the proposed and explored solutions is employing Transfer Learning (TL) techniques.

2.2.3 Deep Q-Networks Algorithm

In (Watkins 1992), the concept of Q-learning is presented as a model-free reinforcement learning algorithm. Like other RL algorithms, it enables an agent to learn optimal strategies through online trial-and-error interactions with an environment. In other words, it operates in the context of a Markov Decision Process (MDP), which comprises the concepts of states, actions, transition probabilities, and rewards. The Markov property dictates that the future state depends solely on the current state and the respective action.

The "Q" in Q-Learning symbolizes the quality of actions in different states, referencing the algorithm's main feature of evaluating and selecting actions that maximize cumulative rewards, which relates to the off-policy nature of the algorithm. An off-policy approach means that the algorithm evaluates and updates a policy different from the one used to take an action, learning from the experiences, and following a policy other than the one it is currently trying to optimize.

This is achieved through Q-Values, also known as action values, which are expected values of future cumulative rewards for each action taken in a given state, represented by $Q(S, a)$, where S is the current state and a is the action. In Q-learning, these values are stored in a table of states and actions.

A branch of the traditional Q-Learning approach is the Deep Q-Networks (DQN) (MNIH; KAVUKCUOGLU; SILVER, 2015), a model-free, online, and off-policy Deep Reinforcement Learning technique. Its main difference lies in the estimation of Q values (expected returns), which, instead of using a table, employ a Deep Neural Network (DNN) as a function approximator, referred to as the Q-function, depicted in Figure 2. This is done through interactions and observations of the environment, resulting in Q-values for each possible action.

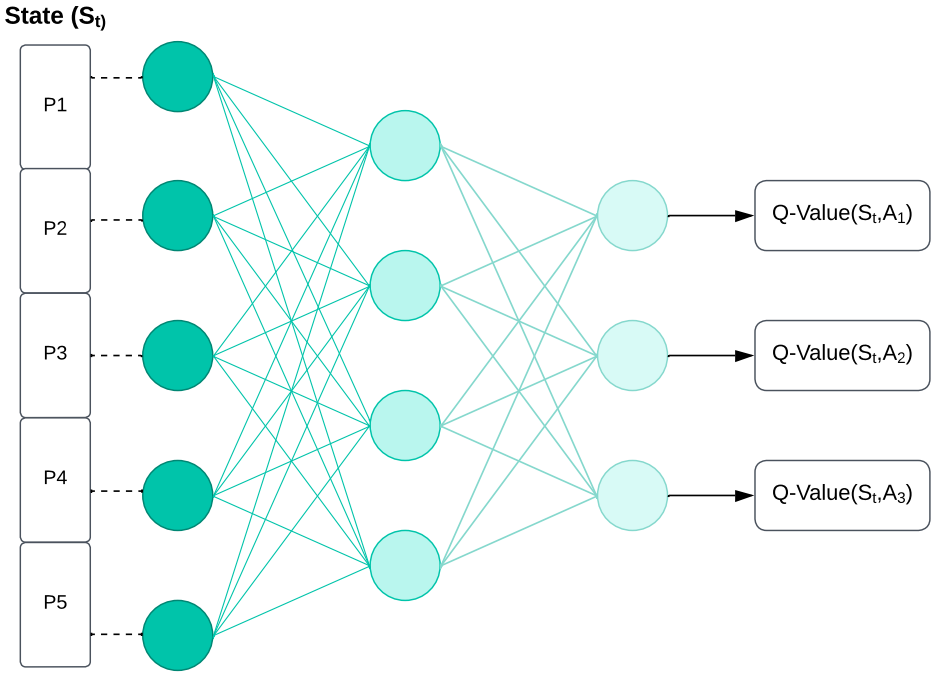

Figure 2 – DQN's neural network representation

Thus, DQN can be applied to problems with continuous observation spaces and large state spaces, learning directly from untreated or unlabeled data and information. In addition to the neural network, DQN introduces the experience replay. Past experiences are stored in a buffer at each iteration and sampled periodically to assess and update the neural network, aiming to break the temporal correlation between consecutive experiences and enhance the stability and efficiency of learning.

Its training process occurs iteratively through continuous interaction between an agent and the environment. The training loop consists of the steps presented on Algorithm 1.

In the initialization phase, the algorithm must initialize its neural network, used to perform the Q-function approximation, a crucial component for decision-making. Additionally, it establishes the experience replay buffer to store and recall past interactions. Essential hyperparameters, such as the learning rate, discount factor, and exploration-exploitation strategy (e-greedy policy), are configured to shape the algorithm's behavior.

As the training initiates, the agent is responsible for interacting with the environment to observe the current state, take actions, and observe the effects caused by its actions

Algorithm 1 Training loop

Initialization Phase

Create Replay Buffer
Create RL Agent
Create Neural Network
Setup Action Environment

Training

Start Episode Loop:
    Reset Environment
    Start Step Loop:
        Observe the Current State and Select Action (e-greedy policy)
        Apply the Action to the System (environment)
        Get Experience: State, Action, Next State, and Reward
        Save the Experience in the Replay Buffer
        Sample a Batch of Experiences from the Replay Buffer
        Train the Online Network with the Experiences and Update Weights
        Softly Update the Target Network with Online Network Weights

Save Policy

on subsequent states. The observation or experience captured by the agent comprises the current state, the action taken, the reward, and the next state.

The deep neural network created for the training process is essential for decision-making. As depicted in Figure 2, it takes as input the environment state S represented by a sequence of numerical attributes and generates a Q-value for each possible action in that state as output. This Q-value indicates the expected cumulative return or reward for the respective action.

To dictate the agent's decision-making process, an e-greedy policy is employed. It defines the probability of taking completely random actions (exploration) or selecting the action with the highest return according to the returns calculated by the Q-function (exploitation).

In each iteration, the selected action is performed, and a related reward is received, composing the agent's experience, which is then stored in the experience replay buffer.

The DQN algorithm employs two neural networks, the online Q-network and the target Q-network. The target Q-network is a copy of the online Q-network, but its parameters are updated less frequently, while the online Q-network is updated in every iteration. This is done to enhance the stability of the learning process by providing more consistent targets.

The experience replay buffer plays a crucial role in the DQN algorithm, allowing the agent to learn from past interactions. Eventually, a batch of past interactions is randomly sampled from the buffer and is processed by the online Q-Network, generating the predicted Q-Values.

During this process, the respective expected target Q-Value is calculated using the

separate target Q-Network, introducing the concept of temporal difference loss. This loss guides the update of the online Q-network's weights through the gradient descent, approximating the predicted and target Q-Values. In other words, the temporal difference loss is the difference between the predicted Q-Value and the target Q-Value.

To enhance stability, the target Q-Network undergoes periodic updates, synchronizing its weights with the current Q-Network. Episodes continue iteratively, with the algorithm adapting its policy based on the learned experiences. This cyclical process persists until a predetermined convergence or termination rule is reached.

The strength of the DQN algorithm lies in its iterative learning approach, continuously optimizing its approximation of the Q function. Through experience replay, it leverages past interactions to mitigate temporal correlations, improving sample efficiency. Ultimately, the resulting Q-Network and its weights constitute the learned agent policy, representing the agent's decision-making abilities within its environment.

2.3 Transfer Learning

In the context of RL, academic literature commonly defines Transfer Learning (TL) as a technique that aims to reuse acquired knowledge from a source domain to benefit the training and learning process of a new target task or domain (TAYLOR; STONE, 2009; PAN; YANG, 2010).

Pan e Yang (2010), states that TL can be relevant, especially in situations where the acquisition of necessary data in the target domain is limited or comes at a high cost. In this way, the context presented in this research aligns with the assertion as we are confronted with a target domain (real-world domain) characterized by its riskiness, complexity, and time-consuming nature of obtaining training data through agent exploration, particularly in the initial training phase. Meanwhile, we employ a source domain (simulation domain), that enables a simpler, safer, and faster collection of the necessary information.

Taylor e Stone (2009) discusses the motivations behind applying transfer learning to RL, such as reducing the amount of training time and data needed for new training and adapting to changes in the environment. The authors also define metrics to measure and evaluate the benefits of TL, such as *jumpstart*, *total reward*, *transfer ratio*, and *time to threshold*. We took these metrics to create our TL evaluation metrics set, which is discussed in Section 5.1.

In our particular context, TL has the potential to facilitate RL training in complex, slow, or resource-intensive environments. Among the expected advantages of applying TL, we anticipate it can improve the accuracy of an RL agent's predictions, reduce the required training time, and even enhance the adaptability of an agent.

Our approach is to generate the source knowledge in a simpler, faster, and controllable environment, such as a simulation environment configured to mirror the real target setting.

Subsequently, the policy produced by the source agent will be transferred to the real-world domain, where the anticipated effects and benefits will be tested. The simplified environment can allow the agent to conduct more interactions in a shorter interval, with no concerns about system safety, integrity, and availability while exploring the existent states, including possible failure states.

In TL, the "knowledge" to be transferred can be represented in various ways, for instance, it is possible to leverage the transfer with sample instances of experiences or observations (state, action, reward), with action-value functions, with complete or partial policies, with task features, and with full task models, depending on the strategy and ML algorithm in use (TAYLOR; STONE, 2009; KAELBLING; LITTMAN; MOORE, 1996).

In our approach, we will conduct the transfer learning using fully trained agent policies established in the training phase, which encompasses a neural network with its balanced weights. This is because we are working with the same task for the source and target domains, while the distinct domains are relatively similar.

The research on TL encompasses a broad range of disciplines and methodologies and is fragmented across diverse domains. Moreover, it continues to attract new investigations, as its potential applications are extensive. We expect that the TL approach will be able to provide benefits within the specific context presented, once again demonstrating its capacity to boost and facilitate the training and learning process.

2.4 NECOS Project

The NECOS project defines a paradigm that introduces a novel business model called "Slice-as-a-Service", capable of providing dynamic, flexible, and efficient orchestration of network slices through the development of innovative technologies for Network Function Virtualization (NFV) and Software-Defined Networking (SDN), what enables the delivery of more secure and efficient services (SILVA et al., 2018).

The concept of a network slice is the partitioning of the cloud network infrastructure to operate independently of other slices. This slicing involves network bandwidth, disk, memory, and processing resources, where various infrastructure resource providers can offer slice parts composed of these resource types. These slice parts are aggregated to compose a complete network slice.

NECOS aims to enable a customer to request customized network slices by providing a description of their needs to the platform. Figure 3 illustrates a crucial part of the architecture proposed in the NECOS project (CLAYMAN et al., 2021). Upon receiving a request from the Tenant Domain, in the Slice Provider, the different types of resources required to create the desired slice are searched in a Resource Marketplace.

The Slice Provider then selects and aggregates all the available different types of resources (slice parts) required to form a single end-to-end customized slice for the ten-

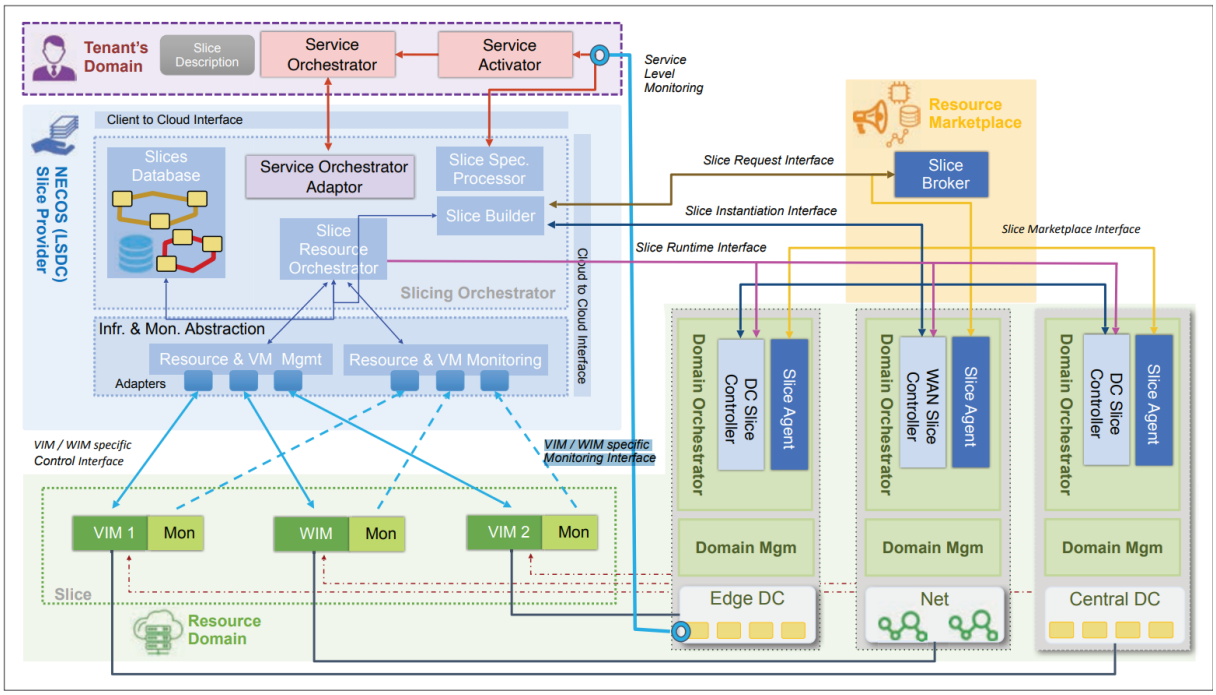

Figure 3 – NECOS architecture

Source: Clayman et al. (2021)

ant. Further, the Slice Resource Orchestrator (SRO), a module of the Slice Provider, is responsible for monitoring the slice and the service through the Infrastructure & Monitoring Abstraction (IMA) module. It dynamically orchestrates the allocated resources based on the performance of the provided service(s) in the slice (CLAYMAN et al., 2021).

Thus, the SRO performs the computing resources management of a computational instance, or network slice, operating as an autoscaling mechanism. The main project does not specify a singular way of implementing resource autoscaling, but in a derived work (REZENDE, 2020) the authors propose and develop an approach for intelligent resource management for the SRO module. In the proposal, it is employed a supervised machine learning technique using the Recurrent Neural Networks (RNN) algorithm.

It involves offline training an ML model from performance and infrastructure metrics collected from the service and its host during an initial period. The training goal is to predict the future value of the service quality metric perceived by the service client in terms of latency, enabling the platform to proactively adjust the allocated resource levels if established performance limits are exceeded.

In REZENDE (2020) perspective, the elasticity provided by the RO is the vertical kind (AL-DHURAIBI et al., 2018), indicating that resource scaling is performed within the slice parts of the host network slice, adjusting computing resources such as network bandwidth, processing power, and memory.

Therefore, the NECOS project stands out as a notable example of ML techniques

applied to cloud computing resource management. It serves as a key reference for our work, aligning with our main proposal, where the potential implications of TL could not only benefit and enhance the NECOS environment but also other similar and related systems.

2.5 Related Work

In this section, we present a review of related works, where we synthesize and analyze existing studies, methodologies, and findings relevant to our research objectives. Through this review, we aim to strengthen the theoretical framework, identify key issues, and highlight areas for further investigation.

In the optimization of service resource management and related contexts, a diverse range of works has explored the application of AI and ML techniques. We have selected some that are related to our approach and have contributed in some way to our research process.

As detailed in Section 2.4, in (REZENDE, 2020), an ML method is employed to provide vertical resource autoscaling to a platform that manages network slices. The employed ML method, from supervised learning, requires a time-consuming pre-execution phase where service operation metrics are collected for offline ML model training. Also, once trained, the model policy remains fixed, and no new data is incorporated to adapt or improve it, leading to potential obsolescence and reduced accuracy in dynamic environments.

Therefore, our research incorporates a Reinforcement Learning (RL) method for a resource orchestrator, where online training enables continuous improvement by updating the policy in real-time and adapting to changing conditions. RL algorithms also support transfer learning, the main subject of this study, potentially eliminating the need for manual data collection and a new model training for each new instance, enabling prompt deployment of pre-trained agents.

The vertical elasticity method has resource allocation limits within each computing instance, and beyond a certain resource demand level, it becomes less cost-effective compared to horizontal scaling. Additionally, this approach may overlook the distributed cloud service's elasticity capabilities, resulting in suboptimal performance and resource waste by solely scaling computational resources within a fixed number of computing instances or nodes. Therefore, to overcome these limitations, in our work, we explore horizontal elasticity.

In (MAO et al., 2016), the authors presented a relatively pioneering proposal for its time. According to the authors, while most real-life resource management problems, at the time, were addressed by human-generated heuristics approaches, they introduced and tested a viable alternative with machine learning by employing a standard RL policy-gradient algorithm coupled with a neural network representing its policy. The algorithm

learns to increase and decrease the capacity of executing computing jobs in a test prototype system. As a result, the technique was demonstrated to be capable of outperforming the compared heuristic strategies in the tested workloads.

In (NOURI et al., 2019), the authors implement a controller powered by RL to scale up and down the resources of a distributed architecture in response to variable demand arrival patterns. They employ the Q-learning RL algorithm for this purpose and formulate a reward function referred to as the "utility function", which enables the system to specify a reasonable trade-off between cost and performance in resource provisioning.

Additionally, they use parameters such as the 95th percentile of system response latency and CPU utilization to categorize the current state of both the infrastructure and the application. The evaluation utilizes the number of violations in the quality of service and the overall cost of the system. They demonstrate that the constructed system was able to reduce violations while minimizing infrastructure costs in certain situations, compared to other control methods without machine learning. However, despite the RL system converging under certain workloads, it does not guarantee stability under all conditions. Our methodology for the RL autonomic resource management draws some inspiration from this work.

In another example of research in resource autoscaling Bitsakos, Konstantinou e Koziris (2018) uses DRL techniques to train the management of VM resources for NoSQL database clusters. They demonstrate that their technique is 1.6 times more effective compared to state-of-the-art techniques using decision trees and standard RL, testing in simulated environments. This work resembles the resource management scenario we propose in our research, where we also employ a DRL algorithm (DQN) to manage the resources of a NoSQL database service cluster (Cassandra). However, their research does not include the study of TL techniques to enhance training and testing in real-world environments.

Transfer learning has been applied in a diverse range of domains, including healthcare, finance, and robotics. A notable example of its application is the work in (APOSTOLOPOULOS; MPESIANA, 2020), where training models for COVID-19 detection in X-ray images were challenging due to the limited size of the available training dataset. However, employing the TL technique boosted the training process, collaborating with the identification of crucial features and achieving significant detection accuracy.

To the best of our knowledge, few studies have explored this specific intersection of deep transfer learning and elasticity orchestration of real-world service environments. However, it was possible to find some cases with significant similarities to our research proposal.

In (WANG et al., 2017), the authors explore DRL to acquire policies for balancing performance and cost within a cloud provider's infrastructure. Although some experiments and training are conducted in a practical cloud environment, the transfer of learning is

employed only between an extremely simple simulator to a more advanced and realistic cloud infrastructure simulator.

Wang et al. (2017) report limitations regarding the extended time required for experiments, which affected the quality of the obtained results. While transfer learning demonstrated some benefits to training in the advanced simulator, its effects on real-world training were not tested or confirmed. In our experience, the extended time for real-world training was also an obstacle, that required some adaptations to the experiments. In contrast to the reference work, we tested the TL effects in a real-world scenario and demonstrated some positive outcomes.

Zhang et al. (2021) leverages a TL approach to facilitate an automated autoscaling solution, that addresses the challenge of determining the adequate resource allocation to meet QoS requirements and minimize resource consumption for video streaming systems. The TL is mainly applied to enhance the system's adaptability to data rate changes.

When dealing with streaming, the model is bound to a defined input data rate, and if the client requests a decrease or increase in the video data rate, the TL algorithm reuses the current ML model for the new rate configuration, avoiding the need to start learning from scratch for every different data rate, thereby reducing adaptation time.

They also employ a similar mechanism to the one we propose in Chapter 3, of a library of previously trained models that can be reused. The strategy demonstrated to be able to reduce resource consumption while ensuring QoS. Our work has similarities but mainly differs in the application scope, as resource management will be applied to cloud computing.

Qiu et al. (2023) defines a framework for deploying and managing RL-based agents in production systems, specifically applied to workload autoscaling in production cloud environments. They employ an alternative approach to TL, aiming to reduce retraining costs and enable faster adaptation to new environments. Additionally, they seek to improve training safety, particularly during the initial exploration phase of online training for production systems.

They demonstrated positive results in terms of adaptation speed and performance stability when compared to rule-based and standard RL approaches. Although they employed a different technique than TL, aiming to be more comprehensive and improve certain aspects, TL remains relevant and can be fundamental for specific scenarios due to its straightforward and less complex implementation and deployment. We seek to corroborate this with our work.

Chapter 3

Proposal

The main proposal of this research is to investigate and validate the application of a transfer learning technique aimed at facilitating and making viable the training process of an RL agent within a real cloud network environment. The objective is to make improvements in terms of quality of performance (cumulative rewards) and time required to train. Specifically, the agent will be trained in tasks related to resource orchestration focusing on horizontal elasticity, within a complex and real environment of a distributed cloud service.

This research proposal is explored and detailed throughout this chapter. It encompasses the presentation of the architecture designed for experimentation and testing purposes, the workflow governing the experiments, and the integration between components.

3.1 Proposed Approach

In response to the main challenges and limitations identified in the realm of RL training in real network and service environments, we present our approach. Our strategy involves the development of pre-trained agents, denoted as source agents, within a more straightforward, controlled, and secure environment. To achieve this objective, this research aims to construct and leverage a simulated representation of the actual environment, serving as a foundation for our experimentation process.

It is expected that the simulation will not only facilitate but also accelerate the RL training process. Therefore, further testing and adjustments of the various configurations and hyperparameters of the RL training can become attainable in a timely manner. This approach may allow the establishment of a hyperparameter set that optimizes the RL simulation training and also provides benefits to the real environment training.

Following this strategy, after training the source agent within the simulated environment, where the operation and conditions mirror the actual environment, the acquired knowledge is transferred. Subsequently, in the real environment, a second agent training

process is initiated. However, for the second training, the agent is loaded/enriched with the knowledge acquired by the simulation source agent.

The transfer of knowledge is expected to enhance the real environment training process, enabling the enriched agent to perform better actions from its first iteration, presenting improvements in initial performance and potentially requiring less time to achieve satisfactory average performance. Such advancements would not be possible in traditional training without prior knowledge. Consequently, significant improvements in both training time and agent performance quality are expected.

The use of a simulation environment will not only facilitate the improvement of agent training in the real environment but will also allow experimentation with the reuse of pre-trained agents in different settings.

To fulfill the objectives of the research and conduct the proposed experiments, a comprehensive test environment will be constructed, encompassing both a simulated and a real environment module.

3.2 Testbed Environments

To conduct knowledge transfer testing within the specified context, it is essential to establish both a target environment, comprising a real-world cloud service cluster implementation, and a source environment, which includes a simulation model corresponding to the actual service.

These environments should have the capability to emulate or simulate the dynamics and interactions typically observed in cloud environments and services. Specifically, they should replicate the relationships between services and clients, encompassing the sending of requests and the reception of the respective responses by the client.

The mentioned service should be distributed, meaning it should be able to operate with a cluster of distributed service instances, referred to as service nodes. This necessity arises from the intention to explore the horizontal elasticity of a cloud service, involving the addition and removal of these service nodes.

As depicted in Figure 4, the architecture designed for the real environment, or target environment, is presented and referred to as the Real Environment Module (REM). The REM includes a distributed service cluster, a request load generator, a node controller, and a sensor client.

Additionally, the RL training module is integrated into the environment to interact with the service, orchestrating its nodes, monitoring its performance, and thus conducting the training of RL agents.

The interaction between the service cluster and the load generator is a key aspect of the experimental environment. The load generator is deployed in an independent machine

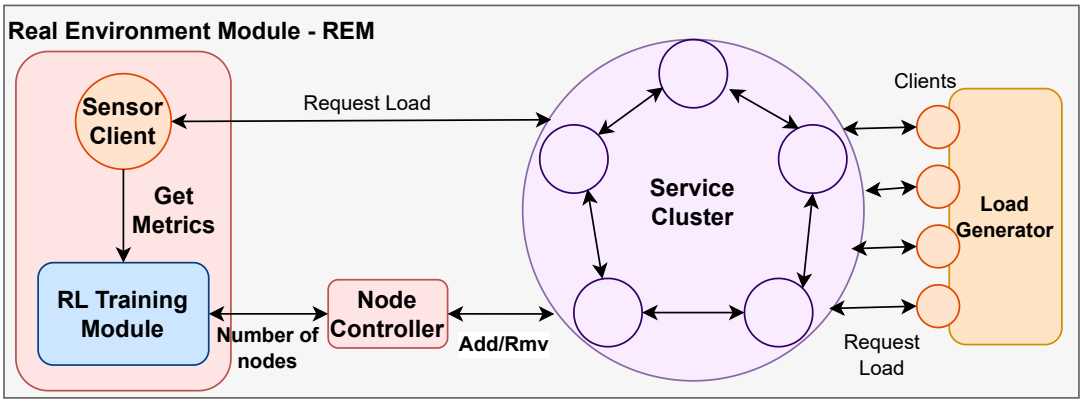

Figure 4 – Real Environment Module (REM) architecture

and generates instances of the stress client sending batches of requests to the cluster machines in a predefined fashion and duration.

The node controller will facilitate the control and monitoring of the nodes in the service cluster, while the sensor client, composed of a client replica, will provide live performance metrics to the RL training. This client should mirror those created in the load generator to ensure its perception of the quality of service is similar to that of all other clients.

In our research, REM will be used to conduct the primary experiments, which consist mainly of the evaluation of the learning transfer. This evaluation consists of comparing the performance of standard RL training (without transfer) with the training enhanced by previously acquired knowledge (with transfer).

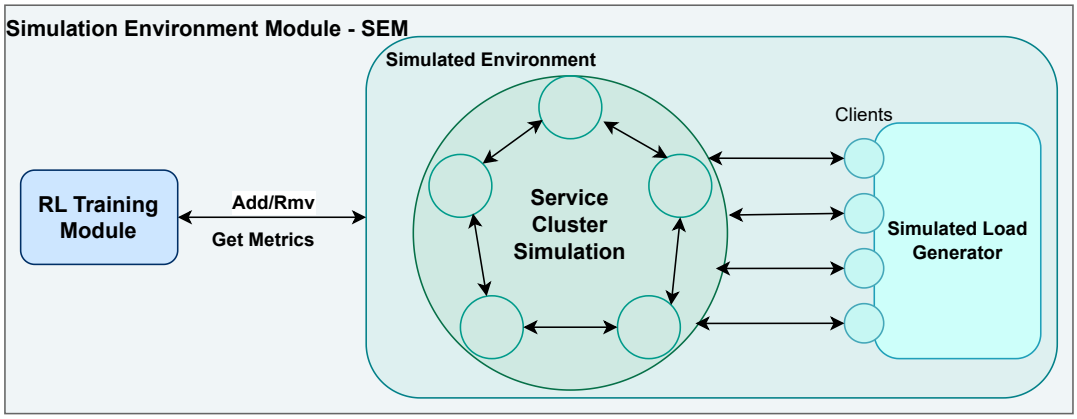

Figure 5 – Simulation Environment Module (SEM) architecture

The Simulation Environment Module (SEM), illustrated by Figure 5, was conceived to provide the ideal conditions for training the RL agents faster and simpler. Its primary goal is to facilitate the training process, while possibly enhancing the training of new agents in the real-world environment through the transfer of learning, by employing a simulation implementation of the target service.

The module encompasses a simulated environment and an RL Training Module. The

simulated service environment replicates the real service cluster environment, consisting of a simulation model of the service cluster and a simulated request load generator for simulating client requests. Intentionally designed to operate like the previously presented real environment, the simulated environment seamlessly integrates with the RL training module. This integration allows the module to execute horizontal elasticity actions on the simulated service cluster and monitor its performance.

The main objective of the simulation module is to offer a simpler, faster, and more consistent environment while preserving the essential characteristics and behaviors of the real service environment we want to control. Consequently, it provides an ideal setting for agents to undergo pre-training in activities related to resource orchestration.

Both the real and simulated environments will be integrated with the RL module, where the RL training will take place, and the agent will interact with the service cluster (real or simulated). Thus, the agent will be able to modify the default state of the cluster, adjusting the number of active nodes, while striving to learn and adapt to the proposed tasks.

3.3 Research Workflow

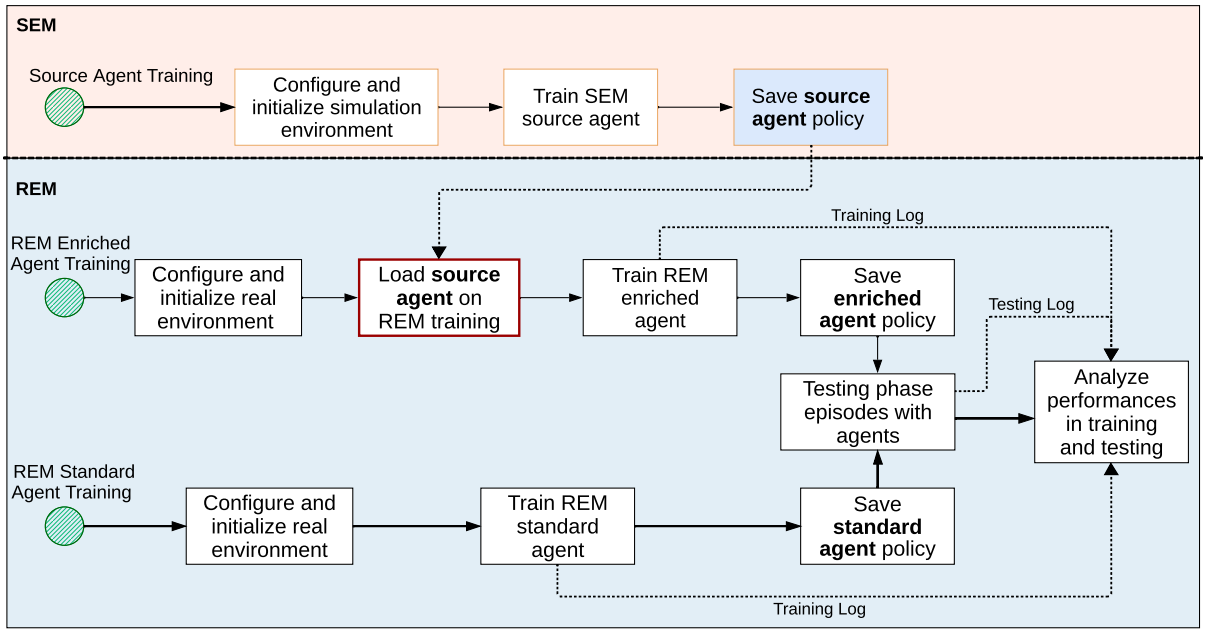

Figure 6 – Experimental research workflow

For the meanings of the TL technique experimentation, it is defined a research workflow with the objective of validating the effects of the knowledge, while assessing whether such transfer yields discernible benefits or not, in the specified context. Thus, in this chapter, the research workflow is described.

The workflow, as illustrated in Figure 6, is divided into two distinct domains: the upper segment, in orange, indicates task execution within the simulated environment (SEM), while the lower segment, with a blue background, designates the real environment (REM) domain. The green circles denote the start point of a new RL training cycle, wherein three distinct training sessions are conducted for each experiment instance or task.

The first stage of the process is the training of the source agent, which takes place in the simulation environment module (SEM). The service cluster and load generator simulations are set up and the necessary hyperparameters are configured, then the RL agent training is initiated, while a request load is generated to the cluster. Upon the conclusion of the training, the source agent's policy is saved and will be reused as the source of the knowledge transfer in the final stage. The simulated environment plays this role because the training process occurs significantly faster in this environment compared to the real environment. Additionally, it provides a simpler and more consistent setting for the exploratory initial operation of an agent.

Before the transfer learning execution, it is required to establish a base agent in the real environment, referred to as *standard agent*, initialized without any prior knowledge. This agent serves as a baseline for control, comparison, and the assessment of transfer learning performance. The standard agent is trained under similar conditions to those of the simulated source agent, meaning it experiences the same level of load stress applied to the service throughout the experiment execution. Moreover, both agents share identical configurations, hyperparameters, reward functions, and states. At the conclusion of both initial training instances, in SEM and in REM, the respective RL agent policy will be stored for reuse.

Thus, at the final stage of the workflow, both environments will be integrated by the transfer learning process. The TL is performed by first transferring the SEM *source agent* policy to the real environment testbed. Then, in the REM, a second training instance will be performed, but this time the RL training algorithm is pre-loaded with the source agent policy in its initiation. In that way, it is expected that the new REM agent, enriched with prior knowledge, be able to demonstrate improvements during its training in terms of performance and time to train.

After completing the training, the policies and logs of two agents will be at our disposal for analysis, both trained in the REM. The first, traditionally trained without transfer (*standard agent*) and the second, trained with TL (*enriched agent*). The performance of these agents is evaluated during a test phase where the exploration rate (e-greedy policy) of the DQN algorithm is set to value zero (0), meaning that no random (exploratory) actions will be executed. This ensures that only optimal decisions are taken based on the respective policy established during the training period.

This setup enables a comprehensive comparison and analysis of the training and testing performances exhibited by these two distinct agents. Through this, we can assess

whether the transfer learning process has the potential to yield significant benefits for the reinforcement learning (RL) training process within its specific context.

3.4 On the Architectural Specification

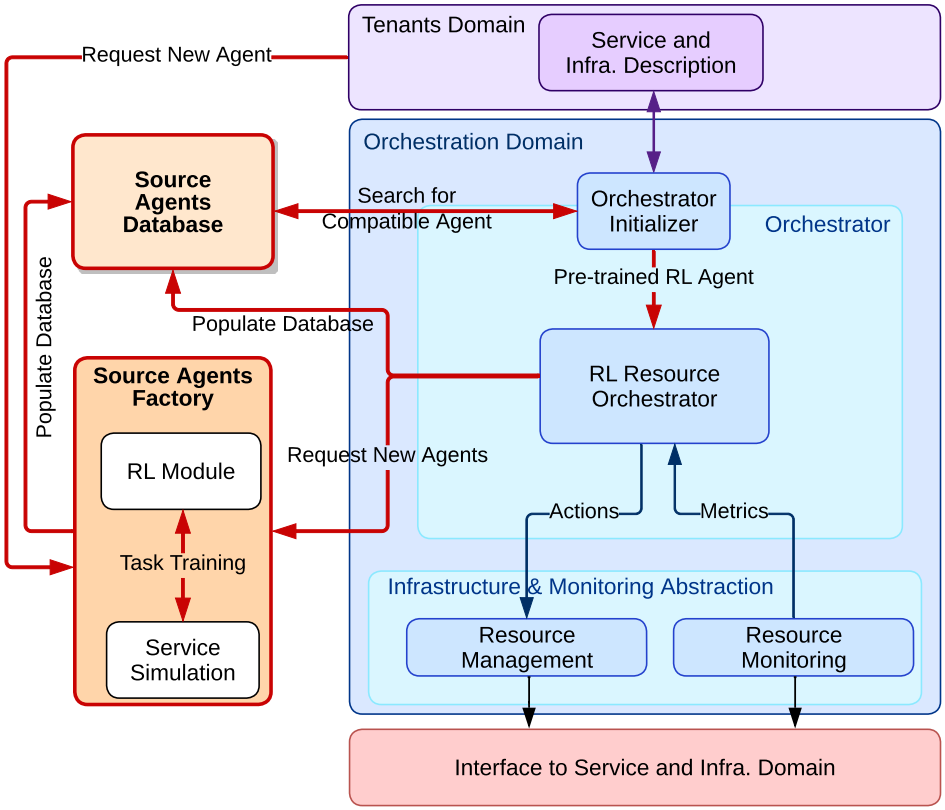

Figure 7 – Proposed theoretical architecture structure

As discussed in Chapter 2, the expected results of our work proposal with TL could enable the idealization of an architecture that integrates the strengths of DRL and TL into its design, producing new functionalities and components. In this section, we present the current theoretical architectural specification to organize our research proposal. The architecture depicted in Figure 7 illustrates this structure, its components, and its relationships, although it has not been practically implemented and tested within the scope of this work. The conceived architecture is based on the NECOS project paradigm described in Chapter 2, as an example scenario. NECOS is a notable case where the features and components envisioned by our research would fit, potentially representing a valuable addition to the existing skill set.

The central component related to our research would be the mechanism responsible for coordinating resource allocation, the RL-driven **Resource Orchestrator** for horizontal elasticity. In this case, it would autonomously manage complete instances of a service with trained RL agents. This orchestrator can either initiate training from scratch, which

may not always be feasible or leverage previously acquired knowledge, providing enough information for the agent to perform reasonably well from the start and gradually improve its performance.

A key contribution of RL and TL techniques in this scenario is the development and maintenance of a repository of pre-trained agents, referred to as **Source Agents Database**. This repository would comprise agents pre-trained in simulated environments, simplified test environments, or even those agents fine-tuned on real operational instances and services (second-level source agents). Similar to a self-service approach, a collection of pre-trained and ready-to-use RL agents would be integrated to serve new infrastructures instantiations.

Thus, when a new resource orchestrator is required, the client must provide the necessary information to the system, including host infrastructure characteristics and a description of the service that will be executed. The **Orchestrator Initializer** should use the information provided to match and select a pre-trained RL agent among the available options in the Source Agents Database. Upon identifying a source agent with a satisfactory level of compatibility, it would be pre-loaded into the intelligent RO. This ensures that the selected source agent not only exhibits compatibility but also matches the new hosted service configuration. The methodology for conducting this agent selection is beyond the scope of this dissertation.

With this approach, the **RO** mechanism will acquire essential specific knowledge from the beginning of its operation, potentially enabling it to deliver satisfactory performance for real and production environments from its initial iteration. Furthermore, it will facilitate the adaptation and optimization of the agent's performance within the newly instantiated environment.

Finally, the running RL agent on the **RO**, continuously learning and adapting to the nuances and peculiarities of the service, structure, and demand, can be backed up, feeding back into the **Source Agent Database**, composing a new agent available for use in new orchestrator instances.

As an additional means of loading and keeping the Agent Database up to date, there could be an additional module that generalizes the role of the simulated environment built in this work, referred to in Figure 7 as the **Source Agents Factory**. This module would be designed to train new source agents for different tasks in distinct services. For this purpose, simulation models for various relevant services need to be prepared and integrated into the RL training module.

In this way, when it is detected a gap in the available agents, indicating the need for a newly trained agent for a specific task or service, either by manual intervention of an operator or automatically, new training sessions would be initiated. The generated agents and policies are then used to populate the Source Agents Database, presenting them as options for future service orchestration instantiations. Thus, some practical aspects in

which the researched techniques can contribute have been highlighted.

Chapter 4

Experimental Testbed

This section describes the main implementations of the project, such as the node controller, load generator, DRL algorithm, and simulations, and the development of the experimentation testbed (REM and SEM), detailing its building and operation characteristics.

To achieve the objectives proposed in this work, detailed in Chapters 1 and 3, it was necessary to design and build a complete testbed capable of providing all the characteristics and features required to conduct the required experiments. The testbed consists of a distributed cloud service environment, a load generator mechanism to stress the service, an RL module to train agents through interactions with the service, and a node controller to make the integration between the MRL and the real-world service cluster, allowing the MRL to trigger the necessary adjusting on the cluster.

For our research, we instantiate the proposed architecture in a specific service to perform the learning and transfer evaluation. We adopted a key-value distributed database service named *Apache Cassandra Database* (Apache Software Foundation, 2019), given that Cassandra is widely employed within cloud network infrastructures and systems, and supports elasticity by automatically adjusting the key space when increasing or decreasing nodes in the cluster. The Cassandra stressing tool *Cassandra-stress* is employed to generate a load of requests in the load generator and the sensor client.

In addition, to complete the experimental environment, it was necessary to build a simulated environment, mirroring the architecture and structure of the real environment. This is mainly composed of a simulation model of a Cassandra service cluster. This simulation also has a stress mechanism that simulates request loads to the simulated cluster. Then, in the same way as in the real environment, the RL module for training agents is configured, adjusted, and integrated into the simulated Cassandra environment. Implementations and integrations are further detailed below.

Machine ID	Description	OS Version	RAM	Disk
1 to 10	Service instances for Cassandra cluster	Ubuntu 16.04	4GB	50GB
11	Node controller system	Ubuntu 16.04	4GB	50GB
12	Load generator	Ubuntu 16.04	8GB	20GB
13	RL module and Sensor Client	Ubuntu 16.04	4GB	100GB

Table 1 – Testbed virtual machines specifications

4.1 Real Environment Module (REM)

The practical test environment of the Cassandra NoSQL database, designed to conduct the experiments, was implemented on an infrastructure of physical servers provided by Faculdade de Computação of Universidade Federal de Uberlândia. On these servers are deployed identical virtual machines orchestrated by OpenStack (RED HAT, INC., 2019). The real environment consists of 4 parts: the Cassandra service cluster, the Cassandra node controller, the reinforcement learning module, and the load generator.

Each Cassandra node is a virtual machine running a Cassandra instance, and the built Cassandra cluster can consist of up to 10 nodes (virtual machines). In addition to the 10 virtual machines that make up the service cluster, the environment has a virtual machine dedicated to the node controller (it coordinates the entry and exit of cluster nodes) and a machine dedicated to the machine learning environment where the DQN agent is trained. Finally, there is a machine dedicated to Cassandra's request load generator.

Through OpenStack, 13 independent machines were virtualized, as shown in Table 1. The first 10 machines have the purpose of creating service instances to compose the Cassandra cluster, running Ubuntu 16.04 operating system, 4GB of RAM memory, 1vCPU, and 50GB of hard disk; one of the machines was assigned to run the node controller system, also running Ubuntu 16.04 operating system, 4GB of RAM memory, 1vCPU and 50GB of hard disk; another machine assigned to run the load generator, running Ubuntu 16.04 operating system, 8GB of RAM memory, 4vCPU and 20GB of hard disk.

The last one, where the entire RL environment will be implemented with the DQN algorithm, and the sensor client will be instantiated, running Ubuntu 16.04 operating system, 4GB of RAM memory, 1vCPU, and 100GB of hard disk.

4.1.1 Cassandra Service Cluster

The Cassandra service cluster is composed of 10 virtualized machines running Apache Cassandra instances and configured to interconnect and intercommunicate and work as a cluster for handling client requests. Although we possess a service cluster of 10 nodes, during the experiments, a maximum of 9 nodes will be used simultaneously, as one will always remain as a backup in case of any unforeseen issues.

The ten virtual machines that constitute the service's cluster were equally configured, using the Bitnami Cassandra virtual machine that comes with the Ubuntu 16 operating system and with Apache Cassandra v3.11.5 already installed and functional (VMware Inc., 2020).

However, before starting the service, in each node/instance the configuration file "cassandra.yaml" was modified, which allows the modification of parameters that determine the initial setup and behavior of the nodes.

In the "cassandra.yaml" file of all cluster machines, the cluster seed nodes are defined. These nodes are responsible for being an initial gateway for cluster formation and the IP of the cluster seed nodes must be provided in the configuration file of all machines so the newly started node knows which seed node to do the first connection.

Seed nodes store information from all nodes that connect to them and send information from all nodes connected to the cluster to newly connected ones, allowing automatic cluster formation as nodes are recognized.

In each instance of Cassandra the following configuration parameters have been modified in the file:

- ❏ cluster_name: 'Cassandra Cluster'
- ❏ seeds: "192.168.0.104,192.168.0.141"
- ❏ listen_address: IP
- ❏ rpc_address: IP
- ❏ endpoint_snitch: GossipingPropertyFileSnitch

The IP indicates the IP address of the machine where the configuration file is hosted. The listen address indicates which IP address the other cluster nodes will use to communicate with this node. The RPC address serves as the listening address for remote procedure calls (RPC).

The snitch endpoint indicates how the cluster topology will be communicated between the nodes, the GossipingPropertyFileSnitch uses the gossip protocol to communicate the data center and rack where each node is located. As this is a basic and local cluster configuration in the university's infrastructure, all nodes are hosted in the same data center and rack.

The service is started on all the nodes and after verifying that the 10 nodes are active and properly connected and communicating in uniqueness as a cluster, the keyspace to be used in the tests can be configured.

The keyspace defines the data types and attributes that will be handled and how they will be stored during stress tests. To carry out the experiments, the use of Cassandra's default keyspace, keyspace1, which has two tables, the standard1 and counter1 table, was

defined. The *standard1* table has 6 columns, one to store a key with a size of 10 bytes and 5 to store data with a size of 34 bytes each, with the 6 columns totaling 180 bytes per action.

Keyspace1 was configured to be used in a multi-node cluster, and its replication factor was modified from 1 to 3, so that each data inserted in any cluster node will be replicated to two more nodes, always keeping 3 copies of each record. After these initial settings, the cluster is ready to receive client requests.

4.1.2 Node Controller

The DQN algorithm requires the ability to interact with the cluster in order to add and remove nodes. However, it would not be possible for it to send the Cassandra commands directly to the service (like `decommission` to remove and `start` to add a node) as there is no way to guarantee that the actions were completed without errors. Additionally, it is complex to know which node is available to be started from the "node pool".

Thus, in the real environment, it was necessary to develop a node control system responsible for orchestrating and monitoring the number of active nodes in the Cassandra cluster. The node controller source code is available in (CUNHA, 2024).

The system must be able to receive the desired value of active nodes and take the necessary actions to adjust the cluster to the desired level, either by adding nodes or removing them.

The script was written in Python and receives as a parameter the value of the nodes that must be active. The algorithm is then responsible for adjusting the cluster until reaching the desired number of nodes, the additions and removals of nodes must follow a specific process to avoid problems in the cluster and data corruption. Only one operation is performed at a time and the algorithm waits for the recommended time to start the next addition or removal operation.

In addition, the algorithm has another important function, it monitors the state of the cluster and keeps two variables updated that are consumed by the RL controller module. The variables are number of active nodes and the number of target nodes. From these values, the RL agent is able to know when operations are in progress in the node controller and how many nodes are active in each training step. All communication between the machines and between the DQN algorithm and the node controller system is done through HTTP REST API requests.

4.1.3 Probe Client (Sensor)

DQN training demands constant input of quality-of-service metrics provided by the Cassandra cluster in terms of response time. To carry out this monitoring and collect the required metrics, it was proposed to perform operations on the Cassandra cluster directly

from the machine where the training takes place, in this way, it is possible to save the response metrics continuously in a log that is read by the algorithm during training to collect the relevant pieces of information.

To carry out operations on the cluster, the stress tool Cassandra-stress is used, which performs several operations per second on the Cluster, according to defined parameters in its command execution. Additionally, the tool allows to register the performance metrics of these operations in a log file.

Thus, the so-called sensor probe client is an instance of the cassandra-stress tool, used in the same way by the load generator to start clients, and its main function is to continuously stress the resources of the Cassandra service during the experiments and record in a log the essential performance metrics obtained every second. It will work as a performance sensor since the test client is identical to the other clients instantiated by the load generator in order to stress the cluster and apply the different defined load patterns.

Every second, the following performance metrics referring to the operations performed are recorded, such as the number of writing and reading operations performed, the average time of these operations, and the 95th percentile of the latency.

The main performance metrics from the probe client log, that will be used in the RL training, are, 95 percentile of read latency, latency mean and latency median. It is expected that these metrics reflect the performance of the Cluster against the request workloads applied by the Load Generator and are suitable for training.

4.1.4 Real Load Generator

The Load Generator, running on a dedicated virtual machine, is responsible for applying a variable request load to the cluster to induce fluctuations in the performance delivered to the clients and will be monitored by the Probe Client. The load generator is built as a Python script, where different instances of cassandra-stress are created in parallel and terminated during the experiment period. This means that multiple instances of cassandra-stress will be running in parallel at certain times, while at other times fewer instances will be running.

Each cassandra-stress instance is created with a preset of a 'threads' parameter, which determines the number of threads used to perform the operations. Therefore, two variables directly influence load generation by the stress tool, the number of client or cassandra-stress instances started by the load generator and the number of threads employed by each of these instances. In general, the instatiated Cassandra clients (sensor client and load generator clients), will employ 10 threads each, executing between two thousand (2000) to eighteen thousand (18000) read requests per second.

An instance of cassandra-stress can emulate a service client, performing operations continuously and in parallel by threads. As the load level generated by this tool is fixed based on the specified number of threads or operations per second, a load generation

script in Python was adapted to allow the creation and deletion of individual instances of cassandra-stress respecting a standard configured load.

4.2 Simulation Environment Module (SEM)

The SEM is deployed and operated on either a personal computer (Windows 11 operating system, 16GB of RAM memory, intel core i7-10510u 10th gen processor, and 256GB of solid-state drive) or at *ClusterUY* supercomputing center (NESMACHNOW; ITURRIAGA, 2019), as part of the established collaborative effort with Universidad de la República, Uruguay (UDELAR). The module encompasses a simulation model of Cassandra service cluster and a simulated request load generator, seamlessly integrated with the RL module (RICHART, 2022).

The simulation environment replicates the real service cluster environment, consisting of a simulation model of the Cassandra service cluster and a simulated request load generator for simulating client requests. The Cassandra simulation model and its accompanying load generator were derived from the model proposed in (DIPIETRO; CASALE; SERAZZI, 2017). However, for our research the model was re-implemented using *Simulink by MathWorks* (MATHWORKS, 2020). This adaptation was required to enhance flexibility in the simulation environment, enabling integrations, including the one with the RL module.

In Simulink, the package *MATLAB Engine for Python* (MathWorks, 2023) is used, enabling the python-built DQN algorithm to call MATLAB functions and interact with the Simulink sandbox, for instance, initiating and restarting the simulation as required. Moreover, this integration allows the Python DQN algorithm to retrieve the simulation cluster state and interact with it by executing actions like addition or removal of cluster nodes.

Similar to the real-world cluster, the simulation model provides the capability to configure the cluster replication factor and consistency level. Additionally, essential performance parameters for RL training, such as throughput and response time, are generated by the simulated service cluster and made accessible for the learning process.

In Figure 8, the implementation of one of the Cassandra nodes using Simulink is illustrated. The simulation is based on queuing theory, and each node includes three queuing engines representing network, CPU, and disk workload. The amount of resources allocated to nodes for each of these three parameters can be customized. Furthermore, each node has input and output queues that interconnect all the Cassandra cluster nodes. These queues allow the node to handle external requests from simulated clients and internode forwarded requests. Additionally, they enable the forwarding of requests between nodes and the transmission of responses to clients or other nodes.

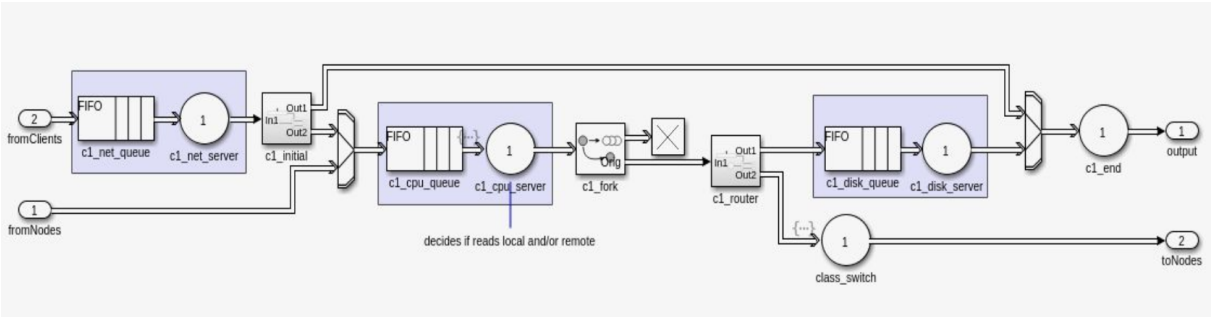

Figure 8 – Simulink Cassandra node representation

4.3 RL Module Design

In this section, we present the details of the RL Module and training, which constitutes the core component of the intelligent resource orchestrator implemented in both experimental environments. The main purpose of the RL module is to deploy the RL algorithm and integrate it into the respective environment, with the goal of training agents to coordinate the addition and removal of service nodes while adapting to varying conditions. Thus, the chosen RL algorithm will be described, including aspects such as state space, action space, reward function, and hyperparameters.

In the previous chapters, a set of requirements was identified for the proposed RO solution, with the objective of mitigating weaknesses identified in the former intelligent RO solution. One of these requirements involves the ability to initiate operations promptly, eliminating the necessity for data collection and labeling. Additionally, the RO should demonstrate the capability to adapt to changes in the environment and, finally, should be compatible with transfer learning, allowing the reuse of pre-trained agents.

The utilization of DQN algorithm is proposed, seeking to meet the specified requirements. As an RL algorithm, DQN performs online training, eliminating the necessity for pre-collected datasets and allowing agents to progressively enhance their decision-making capabilities over time, refining their policies and adapting to variations in the environment. Furthermore, DQN allows for the reuse of pre-trained agents, in this way, the agent can apply its past learning to speed up the process and enhance overall efficiency in orchestrating resources.

4.3.1 Goal and Reward Function

One of the key aspects in implementing a Reinforcement Learning (RL) algorithm, is understanding and defining its main learning goal, that is, how the agent is expected to behave face of the imposed obstacles. The primary concern for our RO lies in optimizing the performance of the orchestrated service (for example, in terms of response time of requests).

While this goal might lead to training an agent able to prioritize quality service performance, it is possible that the agent would take a simpler and more direct approach, which would be employing the maximum available resources all the time. However, when dealing with an actual cloud environment, it is crucial to consider the cost associated with the usage of each unit of resource, billed based on used time or contracted quota.

Thus, relying on the maximum use of available resources is a precarious or even unfeasible solution, emphasizing the importance of finding the minimum resources needed to ensure the desired quality of service. Therefore, the core challenge becomes finding a balance between resource usage and service performance, recognizing that minimizing costs can be just as crucial as optimizing performance in cloud environments.

This balance is particularly crucial in shaping the agent's behavior and the training orientation, in DRL, this is achieved by a thoughtful selection of the *reward function*. The reward function molds the agent's behavior by returning rewards or penalties based on the outcomes of its actions.

In our algorithm, the pursuit of performance quality is embedded in the reward function by only rewarding actions that contributes to compliant service performance. Actions that degrade performance, violating the latency threshold, receive no reward. This is achieved by measuring the difference between the minimum performance threshold defined by the service and the actual observed performance parametrized in the function by the *error* value.

On the other hand, to consider resource usage in the reward function, the reward of the compliant actions depends on the amount of resources used to get that performance, penalizing high resource usage. Thus, the proposed function incorporates the cost of usage through the number of Cassandra *active nodes*. Considering that in a Cassandra cluster, we have a maximum amount of nodes we can activate and also a minimum amount of nodes, we proposed the following function: $MAX\_NODES - ((active\_nodes - MIN\_NODES) * \alpha)$. In this way, when the number of active nodes is higher than the minimum, the reward is decreased. The parameter α controls how much we penalize as the number of nodes increases, and for our experiments, the adopted value was 0.75. Therefore, the reward function can be expressed as:

$$R = \begin{cases} MAX\_NODES - (active\_nodes - MIN\_NODES) \cdot \alpha, & \text{if } error \leq 0 \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

For our specific use case of Cassandra DB, we choose response latency as our performance metric to control. More specifically, we want to avoid the 95^{th} percentile of response latency, as measured by the sensor client, to violate the defined threshold, which is evaluated by the *error* parameter.

4.3.2 Specifications

After defining the reward function, we also need to define the parameters we will read from the environment which will define our states for the RL algorithm as well as the actions to take in the environment.

The parameters selected for our Cassandra use case consist of the following metrics obtained from the sensor client and from the cluster itself: response throughput, which is the number of requests handled per second by the server; ongoing action, if an addition or removal action is currently underway; count of active nodes; median and mean response latency; and delta (difference of actual and target response latency 95^{th} percentile). For the actions, we consider three possibilities: scaling out, achieved by adding a node to the cluster; scaling in, achieved by removing a node from the cluster; and maintaining the current cluster state.

$$[throughput,\ action,\ active\ nodes,\ median\ latency,\ mean\ latency, delta]$$

In addition to the previously presented aspects, two other factors are relevant for the RL module: the epsilon-greedy (e-greedy) policy and the learning rate. The e-greedy policy, which determines the rate balance between explorative and exploitative actions, was set at 0.3 for fixed values. This means that 30% of the actions would be exploratory or random. In other experiment instances, a variable e-greedy value was configured, gradually decaying the e-greedy value from 0.3 to 0.1. The learning rate was maintained at 0.01, a standard value that showed to be suitable within the setup of this work.

Our DQN algorithm (RICHART; CUNHA, 2022), as well as the agent and all required resources, are implemented in *Python 3.6.9 and 3.8.8 (Python Software Foundation, 2021)* employing the *TensorFlow* library (ABADI et al., 2016) versions 2.6.0 and 2.11.0. The implementation follows the process and encompasses the aspects prescribed in the DQN description in Chapter 2.

Finally, the deep neural network used by the DQN algorithm is built as a regular feed-forward deep neural network, that takes the state as input, with hidden dense layers and a final layer that outputs action values (expected cumulative reward) for each possible action.

4.4 Modules Operation Workflow

In this section, the whole operation process is explained, detailing the use of the built modules and how the interaction/intercommunication between its different parts occurs during an experiment in the real environment, exposed in Figure 9. The process is the same in the simulated environment, with some variations.

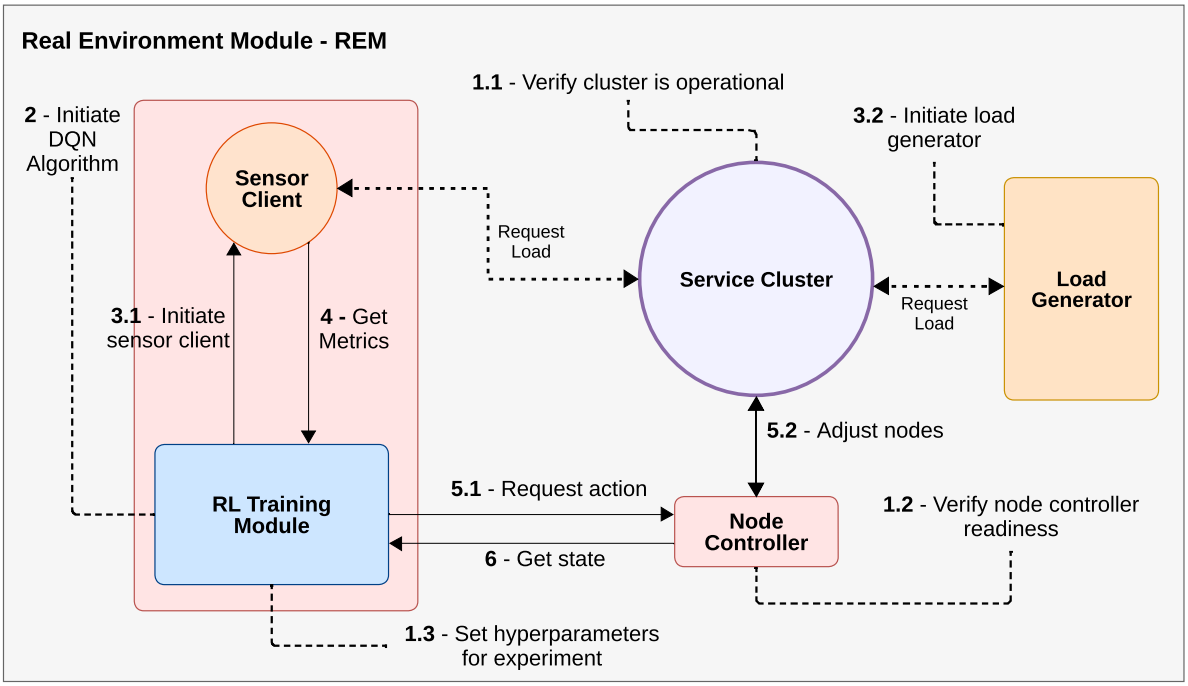

Figure 9 – REM operation workflow

Prior to initiating the RL training process in the REM, it is necessary to verify that the service cluster is active and operational. Additionally, it is necessary to ensure the node controller's responsiveness and confirm its correct initialization and synchronization with the current state of the cluster. Within the algorithm, essential hyperparameters, such as the number of episodes and steps, the performance threshold (target latency), the learning rate value, and the e-greedy value, need to be set.

Subsequently, DQN training can be initiated in the REM, according to the experiment's specific presets. At the beginning of the training episode the DQN algorithm launches the Cassandra sensor client. This sensor client is employed to perform a batch of requests per second for the service cluster while collecting performance metrics related to the request's responses. These service performance metrics are incorporated into the DQN training process, composing the first portion of the DQN agent's observation, which describes the current state of the environment.

Additionally, it is necessary to start the load generator by imposing predefined settings for the specific experiment shaping its load level and pattern. At the same time, the DQN process initiates a continuous interaction with the node controller, through HTTP API requests, receiving the cluster status in terms of the number of active nodes and if there is any action in progress. This information also composes the DQN agent's observation of the environment state.

Furthermore, through the node controller, the DQN process is able to request adjustments in the number of active nodes. At the beginning of each episode, the DQN process

check the state of the service cluster and request the adjustment to the required initial number of nodes.

In every step of a DQN training episode, an action is generated, which means addition, removal, or maintenance. Thus, the DQN process interacts again with the node controller to request the necessary action. The node controller then starts the cluster adjustment process, monitoring its progress until its completion and keeping the DQN algorithm updated on the status.

During the training phase, the agent will explore the environment states, following the e-greedy policy and executing its available actions. For each action taken, a new state is observed, and a reward is granted, indicating whether the action had a positive or negative effect.

Thus, the observations or experiences are employed in the neural network training, gradually updating and optimizing its weights, possibly reaching a convergence point. This process aims to generate a policy capable of efficiently conducting the agent's actions within the specific conditions in which it was trained.

In the simulated environment, the main distinction lies in the cluster and load generator, both created within a simulator. Consequently, within the DQN process, an interface is implemented to initiate the load generator in each episode and execute the DQN actions in the Simulink simulated cluster. Additionally, the role of a sensor client is unnecessary, the service simulation process saves the performance metrics in a log that is accessed and read by the DQN process.

Chapter 5

Experimental Results and Analysis

This chapter describes the experimental segment of the project, demonstrating how the formulated/raised hypotheses were addressed through the proposed experiments and their outcomes.

In Section 5.1, the method employed for experimentation is described, with the aim of achieving the objectives outlined in Chapter 1. It also outlines the strategy and approach used for generating the necessary data, followed by the subsequent analysis and evaluation of the results.

Section 5.2 details the conducted experiments, their primary outcomes, and a discussion concerning these results and their implications regarding the formulated hypotheses and the objectives of the research.

5.1 Evaluation Method

The strategy employed for conducting experiments in the constructed testbed, comprised of SEM and REM, aims primarily to provide the necessary means for the evaluation and validation of the raised hypotheses. This is achieved through the analysis of data and information generated by interactions among the experimental environment modules during tests. The focus of the experiments is on validating a transfer learning technique in the context of cloud service resource orchestration.

Upon completion of the required training phases, as described in Section 3.3, there will be two different agents trained in the real environment: one traditionally trained without transfer (the *standard agent*) and the other trained with TL (*enriched agent*). The performance of these agents is evaluated during a test phase where the exploration policy (e-greedy rate) of the DQN algorithm is set to zero. This ensures that only optimal decisions are taken based on the respective policy established during the training period.

Each training-test set produces information necessary to extract what we refer to as an evaluation metrics set. This set comprises two instances: training evaluation metrics and test evaluation metrics. Both include parameters such as episode reward, moving

average reward in each episode of the test, average node usage in each episode, and the average performance violations for each episode.

- ❏ Episode reward: At each step of a training episode, the agent receives a reward value based on the reward function or the impact caused by the action taken. This parameter represents the sum of all these reward values received during the entire episode.

- ❏ Moving Average Reward: Is the average value of the episode rewards received over the ten most recent episodes.

- ❏ Average Node Usage: The average of active nodes used for the entire episode.

- ❏ Performance Violations: Counts how many steps registered violations to the 95th percentile latency threshold.

These performance metrics from both the training and testing phases enable the measurement of additional parameters. These will be utilized in the result analysis, enabling the comparison and evaluation of the TL training. The key parameters include jumpstart, final performance, and total reward, which were derived from TL evaluation parameters introduced in (TAYLOR; STONE, 2009).

- ❏ Initial Performance: The reward average of the first 10 episodes of a training instance.

- ❏ Jumpstart: The difference in initial performance between two different trainings.

- ❏ Final Performance: The ultimate performance achieved by an agent in the test phase.

- ❏ Total Rewards: The overall reward accumulated by an agent, or the average reward of the entire training.

The extracted evaluation metrics set, from the training and testing of the *standard agent*, is compared to the set of metrics extracted from the TL *enriched agent*. This comparison aims to determine whether the transfer of learning has provided relevant benefits to the training process of an agent within the real-world environment. Also, it enables a identification of specific areas where the benefits were more pronounced or imperceptible.

5.2 Experiments and Analysis

The predominant factor influencing the agent's behavior and its learning is the load of requests applied to the service. The consequences of the applied load can create a challenging and unknown environment for the agent, compelling it to make decisions regarding the level of active resources to maintain minimum service quality and optimal resource utilization. For the subsequent experiments, a fixed load level was employed throughout the duration of each experiment.

For the reward function, we set $\alpha = 0.75$ and MAX_NODES and MIN_NODES to 9 and 3 respectively. We set our performance objective to maintain the 95^{th} percentile of the response time under 300ms.

A relevant aspect of the RL training is the exploration ratio, which determines the rate balance between exploration and exploitation actions. We use the simple *e-greedy* policy, which, given a ratio, randomly chooses between exploration and exploitation when taking an action. It is set between 0.1 and 0.3, meaning that 10% to 30% of the actions would be exploratory (random).

In the SEM phase, 600 to 900 episodes will be performed in each training, and each episode will consist of 150 steps. The episode lasts approximately 25 seconds, resulting in total training times between 3 and 6 hours. While in REM, 50 to 100 episodes will be performed in each training, and each episode consists of 18 to 20 steps. On average, the REM episode lasts between 15 and 25 minutes, resulting in total training times between 18 and 34 hours. For comparison, a single episode of the simulation (SEM) setting, that lasts an average of 25 seconds, if executed in a real environment (REM) could take more than 60 hours, while the whole experiment setting (number of episodes and steps) could lead to more than two years of continuous execution in the real setup.

In the conducted experiments that will be detailed in the sequence, two scenarios are defined, the first one serves as an initial interaction with the methodology, employs a low volume of requests per second to the cluster by the load generator, two thousand (2000) read requests per second for the real-world cluster, and 260 units of load for the simulated service cluster. The objective is to verify the proper operation of the built structure and investigate the competence of the DRL algorithm in effectively training agents in both SEM and REM, in addition to testing the feasibility of the proposed TL technique.

The second scenario increases the complexity of the environment by employing a greater volume of requests per second by the load generator, eighteen thousand (18000) read requests per second for the real-world cluster, and 395 load units for the simulated service cluster. What requires the agent to seek a balance in the cost/performance trade-off. In the first scenario, we will detail each phase of the SEM and REM training in greater detail, while in the second scenario, we will mainly focus on analyzing and evaluating the effects of TL in REM.

Within the experiments in the simulation environment, SEM, only one agent will be

trained, the simulation *source agent*. While in the real environment, REM, two types of agents will be trained, firstly the *standard agent*, which is trained in a traditional way without any previous knowledge, and the *enriched agent*, which is trained after transfer of the learning extracted from the SEM source agent. After the training phase, both agents pass through a testing phase, which assesses the maximum performance the trained agent can provide.

In addition to the final test phase, we will run test phases at different maturity points of the agent training. For instance, in Scenario 2, in additionally to the agent trained for 100 episodes in REM, or fully trained, we will also test the performance of the agent when trained for 50 episodes in REM, which is referred to partially trained agent. Finally, TL agents without any REM training will be tested, referred to as Cold Start test as we employ the SEM *source agent* acquired knowledge (source agent) promptly in the real-world environment, evaluating the performance it is capable of providing without additional training.

5.2.1 Scenario 1

In this initial scenario, a minimal fixed request workload is applied to the service cluster, ensuring it can handle the demand with its minimum capacity (3 active nodes). Thus, the agent must learn how to optimize resource usage by removing nodes from the cluster and sustaining low resource utilization throughout the entire episode. Table 2 shows the parameters that will be used in all training reported in scenario 1.

Table 2 – First scenario training parameters

	Load	Training Steps	Episodes	E-greedy	Duration
SEM Training	260 load units	150	600	0.3 - 0.1	3 hr
REM Trainings	2000 requests/sec	20	50	0.1	18 hr

5.2.1.1 SEM Source Agent

For the first training, conducted in SEM, the experiment was configured with the parameters shown by Table 2, to the training of the *source agent*. Each episode comprises 150 steps, and the total experiment consists of 600 episodes. The e-greedy value, initially set at 0.3, indicates that 30% of the agent's actions are random, promoting exploration of unknown states. This randomness is decreased until reaching 0.1 at the final phase of the training.

The graph in Figure 10 depicts the agent's *moving average reward* evolution and *node usage* for each episode during the training phase. The performance increases up to a maximum threshold, evidencing the agent's decision-making improvement, as it uses fewer resources (service nodes) to optimize rewards.

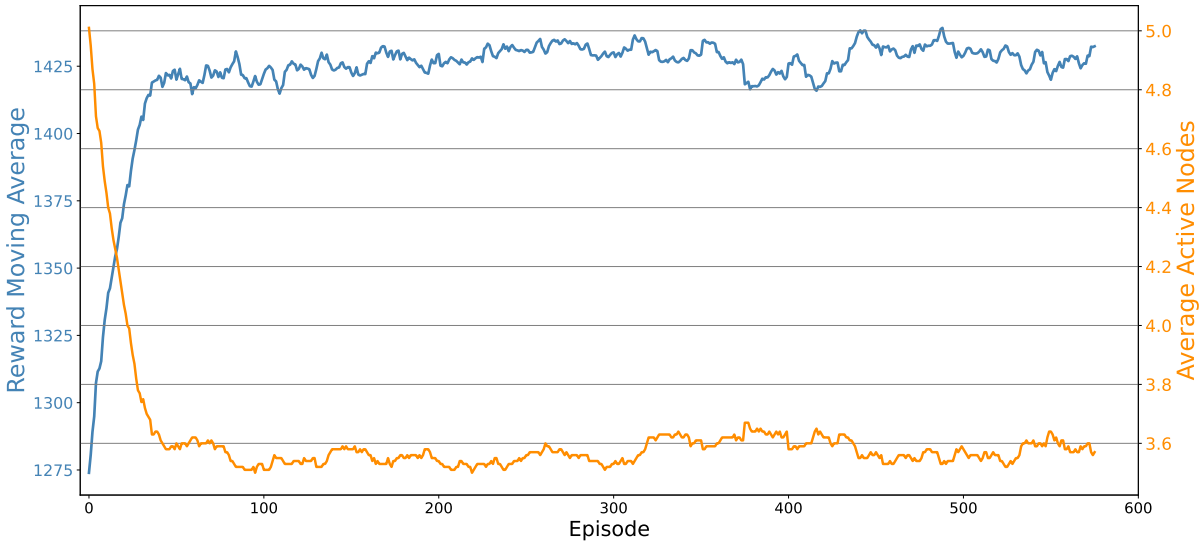

Figure 10 – Simulation training reward moving average and node usage in Scenario 1.

Table 3 presents both the training and testing results. During training, exploratory actions are employed to learn to optimize received rewards in the environment. In testing, the learned policy is exploited, and optimal decisions are made based on the acquired knowledge. The agent's optimal performance achieves an *average reward* of 1478, with an average *node usage* of 3.17 nodes, close to the minimum achievable, and a near zero rate of *violations*, reinforcing the effectiveness of the agent's training and its effort to optimize the received reward by utilizing a smaller amount of resources.

Table 3 – Simulation training and testing results for Scenario 1.

	Avg Reward	Avg Node Usage	Violations
SEM Source Agent Training	1419	3,64 nodes	1 %
SEM Source Agent Testing	1478	3,17 nodes	0,18 %

Figure 11, depicts the source agent behavior on a typical *episode* of the testing phase, plotting the active nodes in use and the 95$^{\text{th}}$ percentile of latency at each step. Initiating with six nodes, the agent adjusts its capacity by removing nodes until it reaches the minimum of 3 nodes, maintaining this configuration until the experiment's end.

The presented results indicate success in the initial training of an agent for a task within the context. In this training, the agent demonstrated the ability to adapt to a load applied to the cluster lower than its response capacity, exploring the attribute of resource usage efficiency. Thus, the SEM source agent for the scenario is established and its knowledge will be utilized in the TL process.

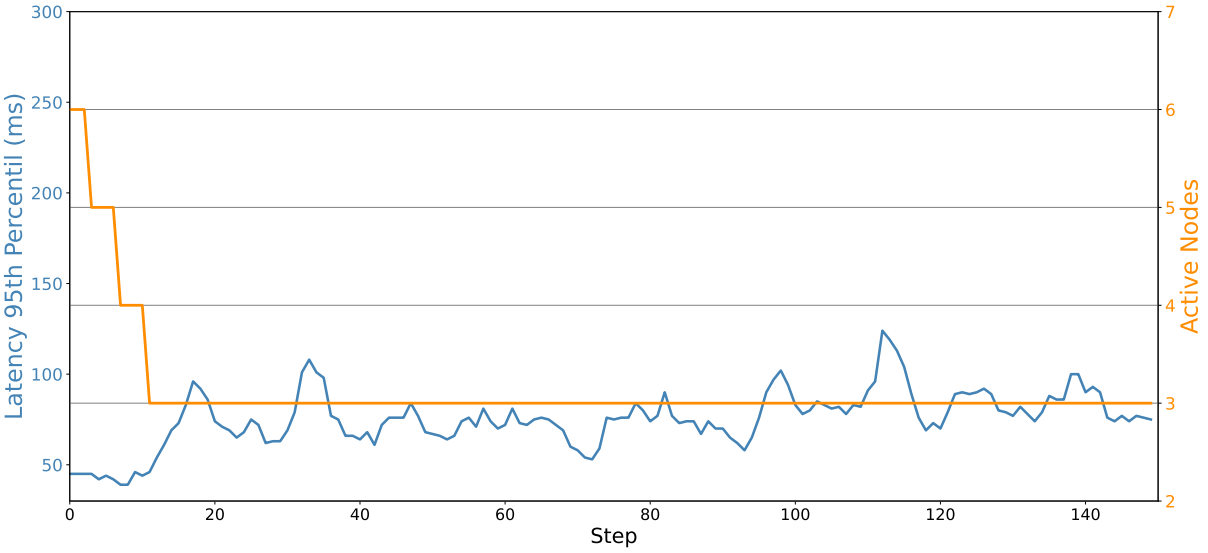

Figure 11 – Latency and active nodes of a typical episode of SEM source agent testing phase

5.2.1.2 REM Standard Agent

The next step is to conduct the standard agent training in the REM. This training should be executed in a similar way as the SEM source agent training. Table 2 outlines the experiment configurations (second row), with adjustments in the number of episodes and steps (50 and 20, respectively) to align with the limitations of the real environment.

The **testing phase** results of the REM *standard agent*, presented in Table 4, reveal an average reward of 187 and a significant improvement in resource usage compared to the training phase, with an average of 3.9 nodes against 4.7. This implies the knowledge acquired in the training phase and exploited in the testing phase in the real-world environment, demonstrating its ability to adapt to the demand received by the service.

Table 4 – Training and testing results of Standard agent of Scenario 1

	Avg Reward	Avg Nodes	Violation	Initial Performance
Training - Standard Agent	175.3	4.7	0%	129
Testing - Standard Agent	187.3	3.9	0%	-

In Figure 12, the **training progress** of the standard agent is depicted in blue, with the moving average reward exhibiting a trend similar to the simulated experiment gradually approaching the maximum achievable reward. The *REM standard agent* is established for the final phase of the experiment to support the evaluation of the TL outcomes.

5.2.1.3 REM Transfer Learning

The final step involves the actual TL execution by conducting an identical training to the previous agent, as detailed in Table 2. The difference lies in the pre-initialization of the training, loading the agent to be trained with knowledge from the SEM *source agent*. The training then proceeds as usual, generating the *enriched agent*.

Figure 12 also shows the moving average reward for the training phase of the REM *enriched agent* in orange.

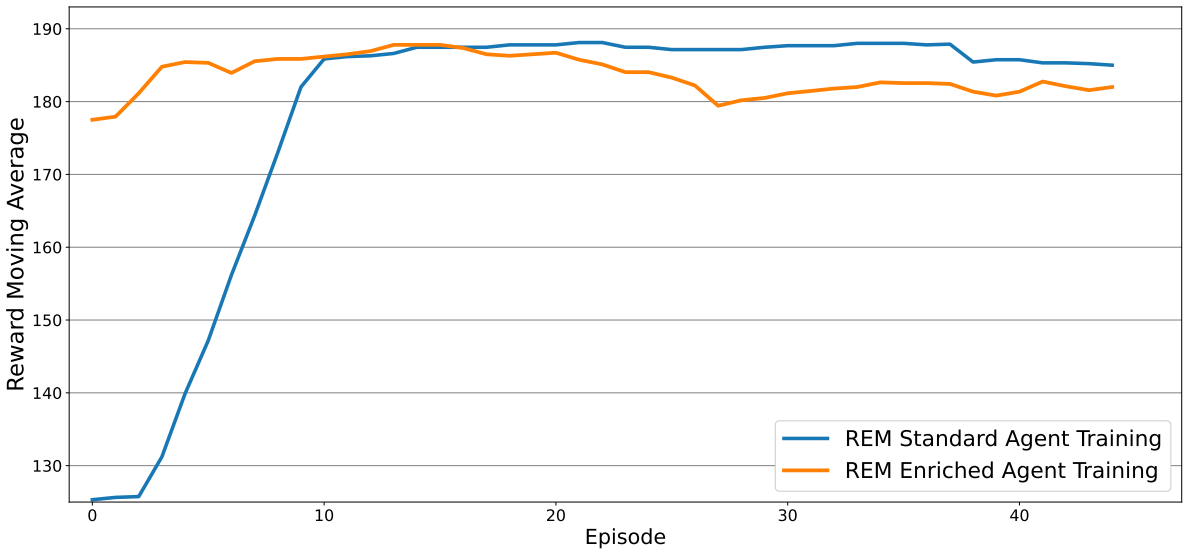

Figure 12 – Standard and enriched agents training reward moving average in Scenario 1.

The **TL evaluation metrics** are collected and analyzed. Key performance results of both training and testing phases, achieved by the *standard* and *enriched agents*, are presented in Tables 5 and 6.

Both agents achieve a close level of reward in training. While the *standard agent* presents a higher and more stable trend after its convergence, the *enriched agent* presents a notable improvement in the initial performance of 39.8%, referred as *jumpstart*. Also, the average reward for the whole training phase is 4.4% greater for the *enriched agent*, and it presents 10.6% lower node usage, as presented in Table 5.

Table 5 – Training results of standard and enriched agents of Scenario 1

	Avg Reward	Avg Nodes	Violation	Initial Performance
Training - Standard Agent	175.3	4.7	0%	129
Training - Enriched Agent	183	4.2	0%	180.4

Figure 13 illustrates the moving average reward of both standard and enriched agents over a 50-episode **testing phase**. Additionally, the figure exhibits the performance of the abovementioned cold start test. In Testing phase, the exploration rate of the e-greedy

policy is set to 0 to ensure the agent makes only optimal decisions by exploiting its knowledge. Both standard and enriched agents deliver high performance, maintaining the average reward close to the maximum and similar resource usage. However, the *enriched agent* exhibits a very stable performance with few variations during the test phase.

Table 6 also presents the performance results of the *cold start* test. The direct reuse of the SEM *source agent* knowledge, with no training in the target real environment, could provide high performance to the cold start agent test. The cold start agent demonstrates an average performance at the same level as the *standard agent*, and very close to the *enriched agent*, with a similar average node usage.

Although it begins the test with a slight disadvantage in reward average, after around 10 episodes, it could reach the optimal performance level achieved by the *enriched agent* in its testing phase, while slightly outperforming the *standard agent*.

Table 6 – Testing results of standard and enriched agents of Scenario 1

	Avg Reward	Avg Nodes	Violation	Initial Performance
Testing - Standard Agent	187.3	3.9	0%	-
Testing - Enriched Agent	188.6	3.8	0%	-
Cold Start Test	187.6	3.9	0%	184.7

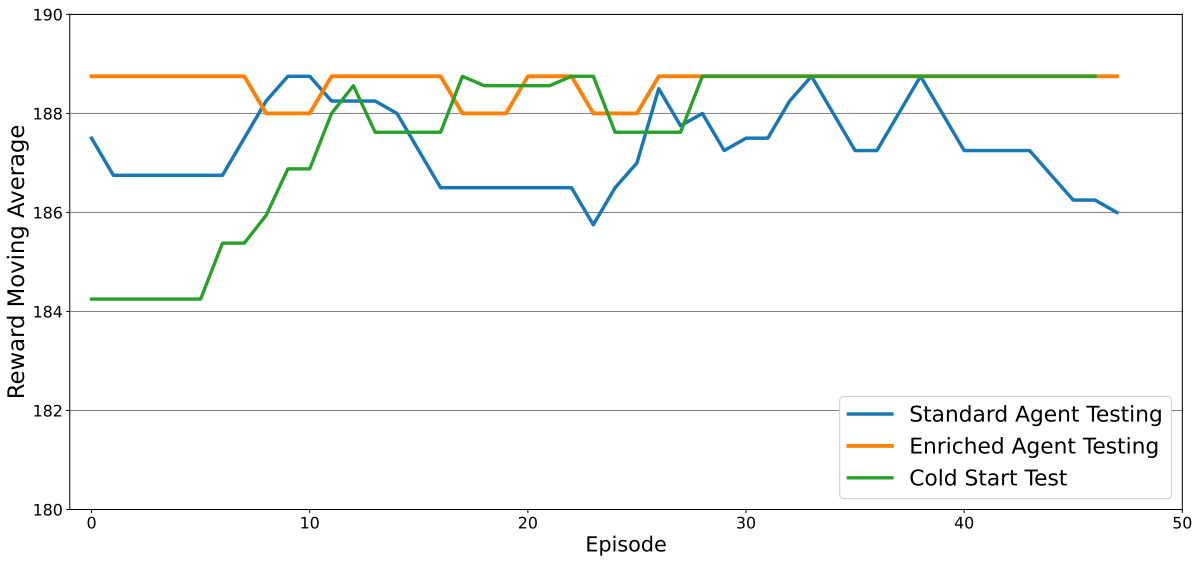

Figure 13 – Standard, Enriched, and Cold Start testing reward moving average in Scenario 1.

5.2.1.4 Discussion of Scenario 1

In summary, in this scenario, we could initially show the proper functioning of the constructed testbed and the effectiveness of training conducted in SEM and REM. Fur-

thermore, even in a simplified task, we could practically demonstrate some of the benefits of TL in this context.

Finally, we demonstrate the possibility of successfully exploiting the knowledge acquired in a simpler environment, without any or little additional training, over the complex target environment (REM).

Because of the nature of the real-world environment, the total time required for the training is significantly longer compared to the simulated environment. Approximately 18 hours were needed to complete the 50 episodes of real training, six times longer than the SEM training for the same task, even with 550 fewer episodes.

This difference reinforces one of the main objectives of the work, which is to positively impact real environment training in terms of time and quality. Thus, successful reuse of knowledge acquired in a fraction of time by the SEM *source agent* could significantly reduce the real environment's prolonged training time.

5.2.2 Scenario 2

In this scenario, a more challenging workload is introduced to test the balance between cost and performance. A fixed moderate request workload is applied to the cluster, so that any lower number of active nodes than the initial setting (6 nodes), is insufficient to handle the demand and maintain service quality.

The agent must learn to optimize resource usage to minimize costs, similar to the previous scenario, but additionally ensuring the quality of the service. Table 7 shows the parameters that will be used in all training reported in scenario 2.

Table 7 – Second scenario training parameters

	Load	Training Steps	Episodes	E-greedy	Duration
SEM Training	395 load units	150	900	0.3 - 0.1	6 hr
REM Trainings	18000 requests/sec	18	100	0.1	34 hr

The training of the SEM source agent for this scenario is conducted over 900 episodes of 150 steps, with a total duration of approximately 6 hours. In this scenario, we will focus on the results and consequences of the REM training.

The complexity of the scenario arises from the behavior of the cluster in the real-world environment, where although the initial 6 nodes are sufficient to handle the workload, the cluster requires a warm-up time to reach its full performance, unable to maintain the required performance at the initial stage. It is expected that as the training progresses, the agent will learn that maintaining the initial resource capacity, although not profitable initially, is the path that should generate the highest accumulated reward over time.

5.2.2.1 REM Agents Training

Table 7 details the REM training configurations, conducted similarly to the previous scenario, but for 100 episodes with 18 steps, due to the higher complexity of the scenario. The key performance results of both agents **training** phase are presented in Table 8 and Figure 14.

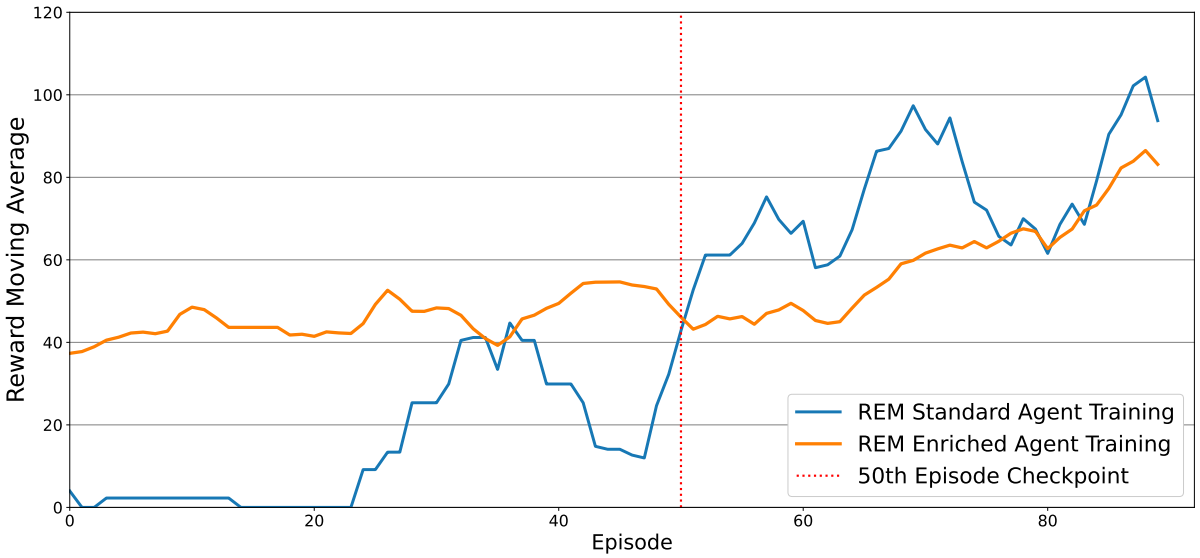

Figure 14 – Standard and Enriched training reward moving average in Scenario 2.

Figure 14 depicts the moving average reward for both agents during the training phase. Despite performance fluctuations during training, the *enriched agent* achieves a higher level of average reward, particularly in the initial stages, demonstrating a 28% improvement of the average reward compared to the REM *standard agent*, as presented in Table 8. In this scenario, the higher performance reflects the enriched agent's superior capacity to avoid performance violations, when the cluster's capacity is insufficient for the demand. Thus, the *enriched agent* demonstrates a notable violation rate 20% lower than the *standard agent* during the training period.

Table 8 – Training phase results (100 episodes) of agents in Scenario 2

Training Phase Results				
	Avg Reward	Avg Nodes	Violation	Initial Performance
Standard Agent	41.9	5.7	70%	4.4
Enriched Agent	53.8	7	56%	43

With initial performance values of 4.4 for the *standard agent* and 43 for the *enriched agent*, the difference, or *jumpstart*, stands at 38.6, representing an improvement of over 800%. This enables the *enriched agent* to initiate training at a performance level that,

without TL, would not have been achieved before the 50th episode, according to its training reward plot evolution.

5.2.2.2 REM Agents Testing:

Figure 15 depicts the moving average reward over a 50-episode testing phase. The performance of the *enriched agent* surpasses that of the standard REM agent, exhibiting a 12% higher and more stable *moving average reward*, along with a 35% reduction in the recorded violation rate. The fully trained agents **testing** phases are presented in Table 9, and Figure 15 presents a plot of the testing phase conducted with the fully trained agents.

Table 9 – Testing phase results of standard and enriched agents, fully trained in 100 episodes, and cold start in Scenario 2

Testing Phase Results				
	Avg Reward	Avg Nodes	Violation	Initial Perform.
Fully Trained Standard Agent	106	6.2	23%	-
Fully Trained Enriched Agent	119	6	15%	-
Cold Start Test	70.4	8.1	38%	50

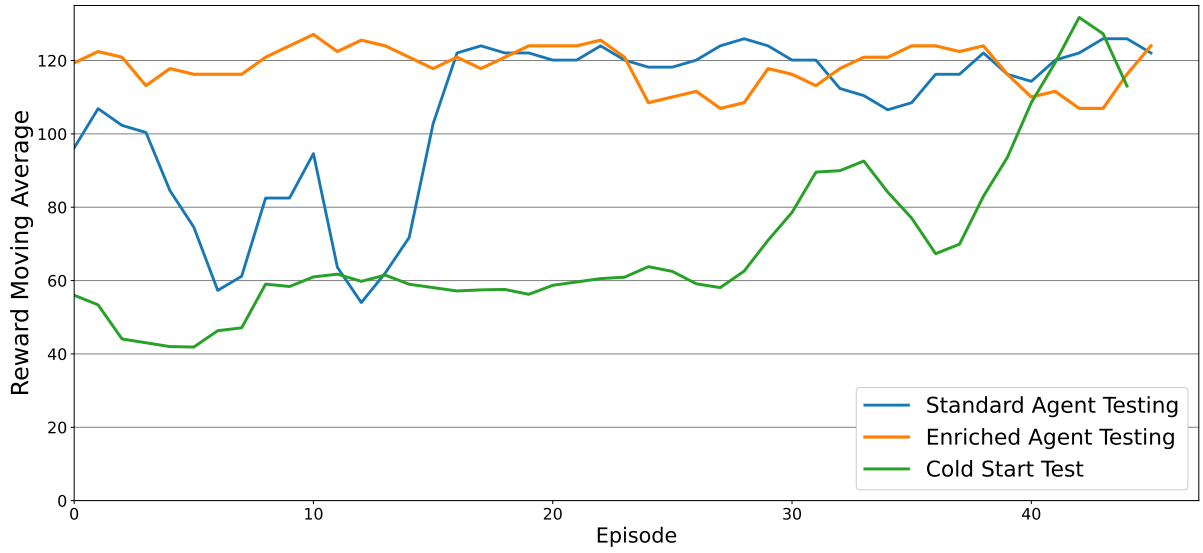

Figure 15 – Fully Trained Standard and Enriched agents testing, and cold start testing reward average in Scenario 2.

The results indicate that the enriched agent, fully trained for 100 episodes, can deliver optimal performance for the presented workload. The average violation rate of 15% is close to the minimum possible, as the initial phase of the experiment will always generate violations regardless of the action taken, because of the warm-up time required for the

cluster to respond appropriately to request loads. On the other hand, the fully trained standard REM agent, despite its high performance and low violation rate, could not match the level of performance and stability exhibited by the enriched agent.

5.2.2.3 Partially Trained Agent Testing and Cold Start Test:

Table 10 – Testing phase results of partially trained agents (trained for 50 episodes) and cold start test in Scenario 2

Testing Phase Results				
	Avg Reward	Avg Nodes	Violation	Initial Perform.
Partially Trained Standard Agent	61.5	6.2	53%	-
Partially Trained Enriched Agent	92.3	6.4	32%	-
Cold Start Test	70.4	8.1	38%	50

In this scenario, an additional testing phase was conducted using the standard and enriched agents trained at the 50th episode checkpoint. This allowed us to examine what performance would be provided by the agents if the training was partially conducted with only 50 episodes. Additionally, we present the cold start test for this scenario, as mentioned in scenario 1, where an agent loaded with knowledge of SEM source agent is tested in REM without training.

Tables 9 and 10 show the testing phase results of the partially and fully trained agents, along with the cold star test result. The partially trained (50 episodes) *enriched agent* achieves a performance close to that of the fully trained (100 episodes) *standard agent*, despite having a considerably higher violation rate. While, we note that, the partially trained *standard agent* could not deliver adequate performance, with an elevated violation rate and average performance far from the optimal performances.

Figure 16 shows a comparison between the results of the cold start test, the test phase of the *standard agent* partially trained (50 episodes), and the fully trained test phase of the *standard agent*. While the cold start test did not surpass the average performance and violation rate of a fully trained standard agent, it did demonstrate that, even without real-world training, it can offer superior performance quality and a reduced violation rate than the partially trained standard agent.

Finally, the *cold start agent* demonstrated to be insufficiently mature to initiate the test immediately providing high performance, as shown in Figure 15. However, it exhibits promising results in this scenario, with an initial moving average reward value of 50, which is a notable achievement for an agent that has not undergone any REM training phase.

Furthermore, it shows an increasing improvement in its average reward throughout the test phase, reaching, at the end of its 50 episodes, a performance level close to the optimal performance of the *enriched agent's* testing phase.

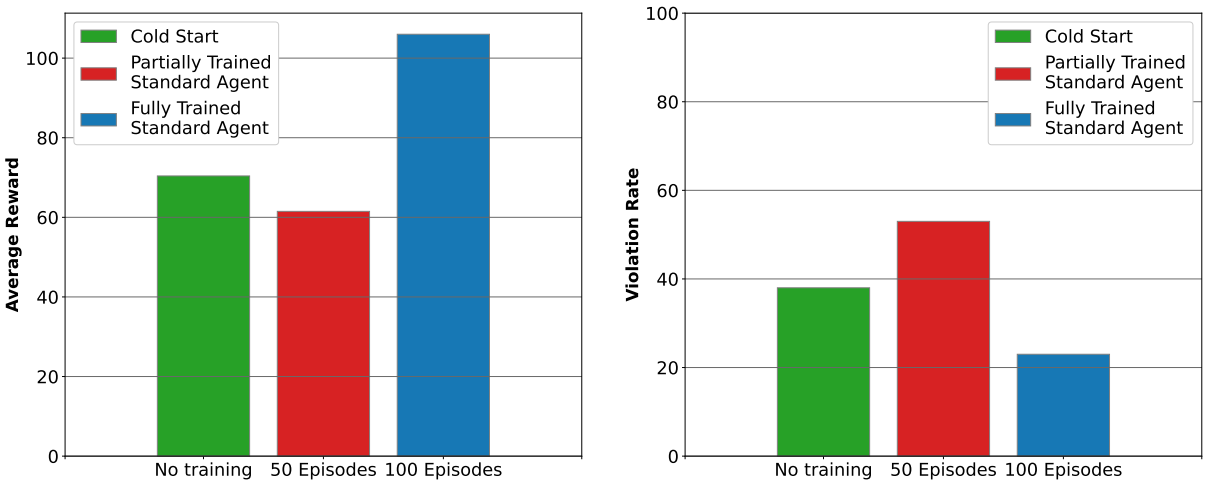

Figure 16 – Average reward and violation rate of cold start test and, partially trained and fully trained standard agents, in Scenario 2.

5.2.2.4 Discussion of Scenario 2

In summary, in this scenario, we could demonstrate the balance between cost and performance achieved in the training and testing phase of agents, achieving efficient performance, especially with the *enriched agent*, by learning to maintain the initial node configuration throughout the episode, even in a counter-intuitive environment.

Furthermore, the obtained results of the TL execution reinforce the initial findings, once again demonstrating its benefits, even more prominent in the initial performance during the training of the enriched agent. And this scenario also demonstrates the pursuit of performance violation prevention.

The test results of partial training reinforce the strength of TL, where even with half of the original training episodes, the enriched agent could provide adequate performance, close to the fully trained standard agent (trained in 100 episodes).

The conducted cold start test reinforces the potential to deploy the *source agent* directly in the real environment with reasonable performance. Possibly resulting in savings of between 18 and 34 hours of training in the workloads addressed in this work, by avoiding the risky training period in critical real-world environments.

In the real world, a simple addition of a node can take hours to be concluded depending on the cluster's configuration, workload, and storage. Therefore, the successful application of such a transfer technique in an environment like this could save days or even weeks of training time. Additionally, it viabilizes the deployment of such kinds of solutions in critical environments that cannot afford the risk of undertaking the initial standard wide exploratory training process.

Chapter 6

Conclusion

In this research, we validate a TL technique for cloud service resource orchestration, aiming to facilitate the training process of DRL agents, reinforcing its viability in critical real-world environments. We built a source environment comprising a simulation of the selected distributed service and transferred the agent's acquired knowledge to enhance the training of new agents in the real environment.

The first objective (G1) was addressed by the designed architecture for the experimental environment and the description of how the modules were constructed in Chapters 3 and 4, along with the successful initial trainings conducted in Chapter 5.

We have effectively demonstrated the feasibility of training RL agents in both real and simulation environments (G2) with the training results outlined and illustrated in Chapter 5, while also proving the capacity of the RL module to interact appropriately with both simulation and real environments by orchestrating the elasticity of the Cassandra service nodes (G3).

Finally, objectives G3 and G4 were addressed by experiments involving transfer learning, where we were able to demonstrate the effectiveness of the technique in providing benefits to the training and testing of RL agents, in the specific context, in terms of training duration, initial performance, and average performance.

6.1 Main Contributions

We were able to demonstrate some of the anticipated benefits of Transfer Learning in RL training over practical real-world scenarios, particularly in its initial performance, showing improvements ranging from 40% to 800% compared to standard training. Additionally, improvements in average performance and savings in training duration for RL agents were attested and reported, addressing the first hypothesis of this research (H1), which proposed that it would be possible to gain benefits by acquiring knowledge in a simple simulation environment and transferring it to the practical environment, enriching the new training.

The balance of the cost/performance trade-off, observed in the training phases of scenario 1 with a 10% reduction in node usage and, in scenario 2, with a 34% decrease in violations, along with the behavior of the agents in test episodes, demonstrating the pursuit of the least cost possible by reducing node usage and the prevent of performance violations by maintaining the needed resource active, validates our second hypothesis (H2), which suggested that it would be possible to train the agent to orchestrate the service node elasticity, aiming for the cost/performance optimization.

Finally, the capacity of TL to accelerate agent maturity or generate agents ready for specific situations was evidenced. This was exemplified in the *cold start* test, where the agent trained in simulation is tested in the real environment without additional training. The cold start agent initially demonstrated reasonable performance, sustaining it throughout the entire experiment. This illustrates the timesaving aspect, where the cold start agent, with only 3 hours of simulation training, operated near or above the performance of the real environment agent, which was additionally trained for 100 episodes in the real environment, saving from 18 to 34 hours required for the training in the REM setup.

The highlighted benefits reinforce the theoretical concepts envisioned in Chapter 3 of the creation of a collection of trained agents, the Source Agents Database, which would be loaded with ready-to-use agents for orchestrating new and different services and environments. The cold start test suggests a promising future for the proposed approach, as we have demonstrated the feasibility of reusing agents in a simplified real-world environment, potentially evolving to a scenario where agents are successfully transferred between different services or tasks. TL enables the orchestration of slice services, meeting the dynamism required for a slice-as-a-service paradigm, right from the beginning of the slice instantiation.

Additionally, another contribution of the work was the implementation and provision of the code for the Cassandra node controller and monitor, which, through API requests, can promptly provide information about the state of the cluster in terms of active nodes and ongoing actions, and automatically adjusts the cluster to the number of nodes sent in the request. All experiment log files, trained agents used and source code are publicly available on (CUNHA, 2024).

6.2 Future Work

In future work, it would be relevant to first enhance the complexity of the conditions imposed on the service cluster to enable the training of more comprehensive agents capable of handling a broader and more realistic range of workload patterns. Additionally, it would be interesting to include trainings with various advanced and sophisticated DRL algorithms, to compare their performances and specific transfer effects.

Furthermore, another aspect to address in future works would be to seek a closer approximation to real-world production environments in terms of scale, storage capacity, and geographical factors. Bridging the gap between experimental and real-world setups can facilitate more accurate evaluations of RL agents and their efficacy in practical contexts.

Finally, the integration of other services into the experimental ecosystem, which demands the training of additional tasks or scenarios, holds potential benefits. This approach allows the exploration of further TL techniques, leveraging knowledge acquired from one service scenario to enhance performance in analogous, yet distinct, service environments.

Transfer learning (TL) is a powerful technique expected to gain increasing relevance, facilitating the implementation of machine learning solutions in diverse and complex contexts.

6.3 Contributions in Bibliographic Production

The work conducted in this dissertation resulted in a paper submitted to the main track of the 42nd Brazilian Symposium on Computer Networks and Distributed Systems (SBRC) in 2024, titled "Transfer of Deep Reinforcement Learning for Cassandra's Elasticity Orchestration".

Bibliography

ABADI, M. et al. Tensorflow: a system for large-scale machine learning. In: **Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation**. USA: USENIX Association, 2016. (OSDI'16), p. 265–283. ISBN 9781931971331.

AL-DHURAIBI, Y. et al. Elasticity in cloud computing: State of the art and research challenges. **IEEE Transactions on Services Computing**, v. 11, n. 2, p. 430–447, 2018. Disponível em: <https://doi.org/10.1109/TSC.2017.2711009>.

Apache, S. F. **What is Cassandra?** 2016. Disponível em: <http://cassandra.apache.org/>.

Apache Software Foundation. **Cassandra Distributed Database**. 2019. Disponível em: <http://cassandra.apache.org/>.

APOSTOLOPOULOS, I. D.; MPESIANA, T. A. Covid-19: automatic detection from x-ray images utilizing transfer learning with convolutional neural networks. **Physical and Engineering Sciences in Medicine**, v. 43, n. 2, p. 635–640, 2020. ISSN 2662-4737. Disponível em: <https://doi.org/10.1007/s13246-020-00865-4>.

ARULKUMARAN, K. et al. Dreep reinforcement learning: A brief survey. **IEEE Signal Processing Magazine**, v. 34, n. 6, p. 26–38, 2017. Disponível em: <https://doi.org/10.1109/MSP.2017.2743240>.

BELOGLAZOV, A.; ABAWAJY, J.; BUYYA, R. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. **Future Generation Computer Systems**, v. 28, n. 5, p. 755–768, 2012. ISSN 0167-739X. Special Section: Energy efficiency in large-scale distributed systems. Disponível em: <https://doi.org/10.1016/j.future.2011.04.017>.

BITSAKOS, C.; KONSTANTINOU, I.; KOZIRIS, N. Derp: A deep reinforcement learning cloud system for elastic resource provisioning. In: **2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)**. [s.n.], 2018. p. 21–29. Disponível em: <https://doi.org/10.1109/CloudCom2018.2018.00020>.

CLAYMAN, S. et al. The necos approach to end-to-end cloud-network slicing as a service. **IEEE Communications Magazine**, v. 59, n. 3, p. 91–97, 2021. Disponível em: <https://doi.org/10.1109/MCOM.001.2000702>.

CUNHA, I. **Cassandra Node Controller API**. Github, 2024. Disponível em: <https://github.com/iansmps/dissertation_repo.git>.

CUNHA, I. R. d. **Construção de mecanismo para suportar a predição de tempos de resposta do Cassandra a partir de métricas de Infraestrutura.** 2019. Disponível em: <https://repositorio.ufu.br/handle/123456789/26441>.

DIPIETRO, S.; CASALE, G.; SERAZZI, G. A queueing network model for performance prediction of apache cassandra. In: **Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools on 10th EAI International Conference on Performance Evaluation Methodologies and Tools**. Brussels, BEL: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2017. (VALUETOOLS'16), p. 186–193. ISBN 9781631901416. Disponível em: <https://doi.org/10.4108/eai.25-10-2016.2266606>.

EL-GAZZAR, R. F. A literature review on cloud computing adoption issues in enterprises. In: BERGVALL-KÅREBORN, B.; NIELSEN, P. A. (Ed.). **Creating Value for All Through IT**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 214–242. ISBN 978-3-662-43459-8. Disponível em: <https://doi.org/10.1007/978-3-662-43459-8_14>.

JENNINGS; STADLER. Resource management in clouds: Survey and research challenges. **Journal of Network and Systems Management**, v. 23, p. 567–619, 2015. ISSN 1573-7705. Disponível em: <https://doi.org/10.1007/s10922-014-9307-7>.

KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement learning: A survey. **Journal of artificial intelligence research**, v. 4, p. 237–285, 1996. Disponível em: <https://doi.org/10.1613/jair.301>.

KEPHART, J.; CHESS, D. The vision of autonomic computing. **Computer**, v. 36, n. 1, p. 41–50, 2003. Disponível em: <https://doi.org/10.1109/MC.2003.1160055>.

KIRAN, B. R. et al. Deep reinforcement learning for autonomous driving: A survey. **IEEE Transactions on Intelligent Transportation Systems**, v. 23, n. 6, p. 4909–4926, 2022. Disponível em: <https://doi.org/10.1109/TITS.2021.3054625>.

LAMPLE, G.; CHAPLOT, D. S. Playing fps games with deep reinforcement learning. **Proceedings of the AAAI Conference on Artificial Intelligence**, v. 31, n. 1, Feb. 2017. Disponível em: <https://doi.org/10.1609/aaai.v31i1.10827>.

LI, R. et al. Deep reinforcement learning for resource management in network slicing. **IEEE Access**, v. 6, p. 74429–74441, 2018. Disponível em: <https://doi.org/10.1109/ACCESS.2018.2881964>.

MAO, H. et al. Resource management with deep reinforcement learning. In: **Proceedings of the 15th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2016. (HotNets '16), p. 50–56. ISBN 9781450346610. Disponível em: <https://doi.org/10.1145/3005745.3005750>.

MARQUES, G. et al. Arcabouço de um sistema inteligente de monitoramento para cloud slices. In: **Anais do I Workshop de Teoria, Tecnologias e Aplicações de Slicing para Infraestruturas Softwarizadas**. Porto Alegre, RS, Brasil: SBC, 2019. p. 56–68. Disponível em: <https://doi.org/10.5753/wslice.2019.7722>.

MATHWORKS. **Simulation and Model-Based Design**. MathWorks, 2020. Disponível em: <https://www.mathworks.com/products/simulink.html>.

MathWorks. **MATLAB Engine for Python: a module to call matlab from python**. MathWorks, 2023. Disponível em: <https://pypi.org/project/matlabengine/>.

MNIH, V.; KAVUKCUOGLU, K.; SILVER, D. e. a. Human-level control through deep reinforcement learning. **Nature**, Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved., v. 518, n. 7540, p. 529–533, fev. 2015. ISSN 0028-0836. Disponível em: <https://doi.org/10.1038/nature14236>.

NESMACHNOW, S.; ITURRIAGA, S. Cluster-uy: Collaborative scientific high performance computing in uruguay. In: TORRES, M.; KLAPP, J. (Ed.). **Supercomputing**. Cham: Springer International Publishing, 2019. p. 188–202. ISBN 978-3-030-38043-4. Disponível em: <https://cluster.uy>.

NOURI, S. M. R. et al. Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications. **Future Generation Computer Systems**, v. 94, p. 765–780, 2019. ISSN 0167-739X. Disponível em: <https://doi.org/10.1016/j.future.2018.11.049>.

OPENAI et al. **Dota 2 with Large Scale Deep Reinforcement Learning**. 2019. Disponível em: <https://doi.org/10.48550/arXiv.1912.06680>.

PAN, S. J.; YANG, Q. A survey on transfer learning. **IEEE Transactions on Knowledge and Data Engineering**, v. 22, n. 10, p. 1345–1359, 2010. Disponível em: <https://doi.org/10.1109/TKDE.2009.191>.

Python Software Foundation. **Python**. 2021. Disponível em: <https://docs.python.org/3.8/reference/>.

QIU, H. et al. AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems. In: **2023 USENIX Annual Technical Conference (USENIX ATC 23)**. Boston, MA: USENIX Association, 2023. p. 387–402. ISBN 978-1-939133-35-9. Disponível em: <https://www.usenix.org/conference/atc23/presentation/qiu-haoran>.

QU, C.; CALHEIROS, R. N.; BUYYA, R. Auto-scaling web applications in clouds: A taxonomy and survey. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 51, n. 4, jul 2018. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3148149>.

RED HAT, INC. **Introdução ao OpenStack**. 2019. Disponível em: <https://www.redhat.com/pt-br/topics/openstack>. Acesso em: 11 jul. 2019.

REZENDE, A. G. P. **Orquestração de cloud-network slices orientada à predição de métricas de serviço a partir do monitoramento da infraestrutura.** Dissertação (Mestrado), 2020. Disponível em: <https://doi.org/10.14393/ufu.di.2020.3053>.

RICHART, M. **Cassandra Simulink**. GitLab, 2022. Disponível em: <https://gitlab.fing.edu.uy/mrichart/cassandra_simulink>.

RICHART, M.; CUNHA, I. **Cassandra Elastic DQN**. GitLab, 2022. Disponível em: <https://gitlab.fing.edu.uy/mrichart/cassandra_elastic_dqn>.

SCHMIDHUBER, J. Deep learning in neural networks: An overview. **Neural Networks**, v. 61, p. 85–117, 2015. ISSN 0893-6080. Disponível em: <https://doi.org/10.1016/j.neunet.2014.09.003>.

SILVA, F. S. D. et al. Necos project: Towards lightweight slicing of cloud federated infrastructures. In: **2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)**. [s.n.], 2018. p. 406–414. Disponível em: <https://doi.org/10.1109/NETSOFT.2018.8460008>.

STADLER, R.; PASQUINI, R.; FODOR, V. Learning from network device statistics. **J. Netw. Syst. Manage.**, Plenum Press, USA, v. 25, n. 4, p. 672–698, oct 2017. ISSN 1064-7570. Disponível em: <https://doi.org/10.1007/s10922-017-9426-z>.

TAYLOR, M. E.; STONE, P. Transfer learning for reinforcement learning domains: A survey. **J. Mach. Learn. Res.**, JMLR.org, v. 10, p. 1633–1685, dec 2009. ISSN 1532-4435. Disponível em: <https://dl.acm.org/doi/10.5555/1577069.1755839>.

VMware Inc. **Bitnami package for Apache Cassandra**. 2020. Disponível em: <https://bitnami.com/stack/cassandra>.

WANG, Z. et al. Automated cloud provisioning on aws using deep reinforcement learning. **ArXiv**, abs/1709.04305, 2017. Disponível em: <https://doi.org/10.48550/arXiv.1709.04305>.

ZHANG, L. et al. Autrascale: An automated and transfer learning solution for streaming system auto-scaling. In: **2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [s.n.], 2021. p. 912–921. Disponível em: <https://doi.org/10.1109/IPDPS49936.2021.00100>.

ZHU, Z. et al. Transfer learning in deep reinforcement learning: A survey. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, v. 45, n. 11, p. 13344–13362, 2023. Disponível em: <https://doi.org/10.1109/TPAMI.2023.3292075>.