Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Intelligent autoscaling in Kubernetes: the impact of container performance indicators in model-free DRL methods

TOMMASO PRATURLON

Stockholm, Sweden, 2023

Intelligent autoscaling in Kubernetes: the impact of container performance indicators in model-free DRL methods

TOMMASO PRATURLON

Master's Programme in ICT Innovation, Cloud and Network Infrastructures
Date: October 13th, 2023

Examiner: György Dán, gyuri@kth.se
Supervisor at KTH Royal Institute of Technology: Feridun Tütüncüoglu, feridun@kth.se
Supervisor at Aalto University: Jukka Manner, jukka.manner@aalto.fi

External advisor: Alex Sundström, asundstrom@spotify.com, Spotify AB

Swedish title: Intelligent autoscaling in Kubernetes: påverkan av containerprestanda-indikatorer i modellfria DRL-metoder

Abstract

A key challenge in the field of cloud computing is to automatically scale software containers in a way that accurately matches the demand for the services they run. To manage such components, container orchestrator tools such as Kubernetes are employed, and in the past few years, researchers have attempted to optimise its autoscaling mechanism with different approaches. Recent studies have showcased the potential of Actor-Critic Deep Reinforcement Learning (DRL) methods in container orchestration, demonstrating their effectiveness in various use cases. However, despite the availability of solutions that integrate multiple container performance metrics to evaluate autoscaling decisions, a critical gap exists in understanding how model-free DRL algorithms interact with a state space based on those metrics. Thus, the primary objective of this thesis is to investigate the impact of the state space definition on the performance of model-free DRL methods in the context of horizontal autoscaling within Kubernetes clusters. In particular, our findings reveal distinct behaviours associated with various sets of metrics. Notably, those sets that exclusively incorporate parameters present in the reward function demonstrate superior effectiveness. Furthermore, our results provide valuable insights when compared to related works, as our experiments demonstrate that a careful metric selection can lead to remarkable Service Level Agreement (SLA) compliance, with as low as 0.55% violations and even surpassing baseline performance in certain scenarios.

Keywords

Cloud computing, container autoscaling, resource optimisation, Deep Reinforcement Learning, Actor-Critic, Kubernetes, service mesh

Sammanfattning

En viktig utmaning inom området molnberäkning är att automatiskt skala programvarubehållare på ett sätt som exakt matchar efterfrågan för de tjänster de driver. För att hantera sådana komponenter, container orkestratorverktyg som Kubernetes används, och i det förflutna några år har forskare försökt optimera dess autoskalning mekanism med olika tillvägagångssätt. Nyligen genomförda studier har visat potentialen hos Actor-Critic Deep Reinforcement Learning (DRL) metoder i containerorkestrering, som visar deras effektivitet i olika användningsfall. Men trots tillgången på lösningar som integrerar flera behållarprestandamått att utvärdera autoskalningsbeslut finns det ett kritiskt gap när det gäller att förstå hur modellfria DRL-algoritmer interagerar med ett tillståndsutrymme baserat på dessa mätvärden. Det primära syftet med denna avhandling är alltså att undersöka vilken inverkan statens rymddefinition har på prestandan av modellfria DRL-metoder i samband med horisontell autoskalning inom Kubernetes-kluster. I synnerhet visar våra resultat distinkta beteenden associerade med olika uppsättningar mätvärden. Särskilt de set som uteslutande innehåller parametrar som finns i belöningen funktion visar överlägsen effektivitet. Dessutom våra resultat ge värdefulla insikter jämfört med relaterade verk, som vår experiment visar att ett noggrant urval av mätvärden kan leda till anmärkningsvärt Service Level Agreement (SLA) efterlevnad, med så låg som $0,55\%$ överträdelser och till och med överträffande baslinjeprestanda i vissa scenarier.

Nyckelord

Cloud computing, container autoscaling, Optimering av resurser, Deep Reinforcement Learning, Actor-Critic, Kubernetes, service mesh

Acknowledgements

I consider myself very lucky to be in the privileged position which allowed me to complete my master's studies and live the university experience the way I did. I am grateful to all the people who support and encourage me, and because I have always been aware of their presence in my life, and perhaps as a little reminder for me, I am also thankful to myself for having had the courage to embrace the opportunities that I recognised, despite my initial uncertainties about my ability to rise to those occasions. Here, I want to acknowledge the support I received from the people who, in different ways, have been involved in this project and express my gratitude towards them.

I want to thank Feridun Tütüncüoglu and György Dán, my supervisor and examiner at KTH Royal Institute of Technology, for their precious comments on my work, and their availability to discuss my ideas. I also thank Jukka Manner, my supervisor at Aalto University, for his approval of the thesis topic since the first moment.

I want to thank Alex Sundström and Mickael Knutsson, my supervisor and manager at Spotify, and Erik Lindblad, for their support and expertise, and for everything else they provided to make my experience at the company the most enjoyable. I must thank Anders Hagman, for his energy and excitement for this project since our first conversation, his faith in proposing the project to me, and his effort in making it happen.

Most importantly, I want to thank my family and Silvia for the love I never felt missing and their support on every occasion despite our distance, and for all other things that words simply cannot describe. And I thank my friends, close and distant, for the incredible and sweet time spent together in these past two years, possibly the best time of my life so far.

Contents

List of Figures

List of Tables

Acronyms

A2C	Advantage Actor Critic
A3C	Asynchronous Advantage Actor Critic
ANN	Artificial Neural Network
API	Application Programming Interface
AR	Auto-Regressive
ARIMA	Autoregressive Integrated Moving Average
Bi-LSTM	Bi-directional Long Short Term Memory
CPU	Central Processing Unit
DDPG	Deep Deterministic Policy Gradient
DNS	Domain Name System
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
GCP	Google Cloud Platform
GKE	Google Kubernetes Engine
GPU	Graphical Processing Unit
gRPC	Google Remote Procedure Call
HPA	Horizontal Pod Autoscaler
HTM	Hierarchical Temporal Memory
HTTPS	Hyper Text Transfer Protocol Secure
HTTP	Hyper Text Transfer Protocol
IaaS	Infrastructure as a Service
IP	Internet Protocol
K8s	Kubernetes
LLC	Last-Level Cache
LSTM	Long Short-Term Memory
MDP	Markov Decision Process
ML	machine learning
MSA	microservice architecture
OS	Operative System

PaaS	Platform as a Service
QoS	Quality of Service
RAM	Random Access Memory
RL	Reinforcement Learning
RPS	requests per second
RTT	Round Trip Time
SaaS	Software as a Service
SARSA	State-Action-Reward-State-Action
SGA	Stochastic Gradient Ascent
SLA	Service Level Agreement
SLI	Service Level Indicator
SLO	Service Level Objective
SSH	Secure Shell
SCTP	Stream Control Transmission Protocol
TD	Temporal Difference
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol / Internet Protocol
UDP	User Datagram Protocol
vCPU	virtual CPU
VM	Virtual Machine
VMs	Virtual Machines
vMemory	virtual Memory
VPA	Vertical Pod Autoscaler

Chapter 1

Introduction

This research is motivated by the need to examine in detail how the performance of model-free DRL algorithms in container autoscaling is affected by the state space definition. In fact, even though the literature presents various DRL approaches that integrated multiple container performance indicators in the state space definition to take the autoscaling action, there is an absence of evaluation regarding how these algorithms interact with a state space that relies on those metrics.

1.1 Background

Cloud computing has become the driver of growth and success of many IT enterprises because of the efficiency and speed of innovation it enables in highly dynamic economical and technological environments such as those we live in today [55]. In particular, cloud computing leverages virtualisation technologies such as software containers to accommodate the requirements of high flexibility and scalability of modern applications [53]. On top of that, container orchestrators like Kubernetes have emerged as a key solution for efficiently managing containerised applications at scale [61]. However, optimising the performance of such orchestrators is a complex task due to the dynamic nature of containerised environments and the diverse requirements of modern implementations [69]. Recently, to solve such sophisticated problems, Reinforcement Learning (RL) techniques have

been employed with various approaches, but the research in this area is still ongoing [71].

Cloud computing refers to the delivery of on-demand computing resources over the Internet, including servers, storage, databases, networking, software, and analytics [65]. In practice, companies adopting the cloud do not build and maintain their own networking infrastructure, rather, they rent the equipment they need from cloud providers, which sell their computers and networking infrastructure as a service over the Internet. A core feature of the cloud, known as *elasticity* or *scalability*, is its ability to handle increasing demand for resources by efficiently allocating new resources where needed.

One of the main technologies that enable the cloud computing model to work is virtualisation [53]. In the past decade, since the launch in 2013 of Docker, an open-source project for Operative System (OS) and application virtualisation, software containers have been increasingly integrated with the use of Virtual Machines (VMs) to manage and deploy applications in the cloud [11]. Essentially, software containers can be described as lightweight versions of virtual machines running a single application each. Two advantages of containers with respect to VMs are their resource efficiency, because they leverage a higher level of abstraction, and their faster deployment and scalability, since their smaller size enables faster operations in a cloud platform.

A common use case for containers is to logically decompose an application into services with very specific tasks, and then individually package such components into containers that can communicate with each other [69]. This practice is called microservice architecture (MSA) and allows applications to adjust the resources for individual services with a fine granularity, which reduces the scaling cost compared to conventional scaling of more resource-intensive VMs. However, when numerous components are needed, many containers have to be deployed, maintained and orchestrated, and these tasks are not easy to handle. Currently, to manage these duties, container orchestration tools such as Kubernetes [40] are widely adopted [61].

Specifically, a crucial feature of Kubernetes is its HPA, a functionality which automatically scales an application based on given threshold policies. This component is extremely useful when, for example, a container becomes overloaded with incoming requests from another

external service and it is unable to maintain the standard Quality of Service (QoS). In order to prevent the container from becoming inadequate to process all requests, Kubernetes' HPA creates one or multiple copies of the same container, called *replicas*, and balances the incoming load among these. Despite its advantages and popularity, the way in which the HPA manages the scaling process is an area that currently needs optimisation [50].

One promising approach to address the scaling challenge is to apply decision-making models such as RL algorithms to the autoscaler [71]. Specifically, this family of algorithms is well suited for problems that can be framed as MDP, which is a mathematical framework used to describe situations involving an *agent* that is making decisions based on the current state of the *environment* in which it is inserted. Based on the *state space* definition, which is the set of all possible states that an agent can perceive and interact with within an environment, the algorithm tries to learn which actions provide the highest probability of reaching a given objective, formulated as a reward function. In particular, two different approaches that can be used to teach the agent the desired behaviour are model-free and model-based RL. In model-free RL, the learning agent has no prior knowledge about which actions lead to the highest reward, instead, it has to discover these by trying them directly in the environment and learning from its mistakes [57].

In the context of container autoscaling, model-based RL can lack generality in highly dynamic situations such as MSA applications, which could cause the simulation of the environment to be invalid [71]. Therefore, the latest resolution to process the dynamically changing workloads in MSA has been model-free RL algorithms, with a focus on SLA assurance and resource efficiency optimisation [48], [66], [67]. However, the present studies on the autoscaler only partially consider the issue of multiple performance metrics to evaluate the scaling action. In fact, all these studies are concerned with creating an end-to-end solution that can autonomously scale microservices by selecting the optimal policy only based on a couple of metrics.

In this respect, when it comes to balancing trade-offs between different metrics, the authors of [71] conclude that Actor-Critic, a DRL hybrid architecture combining value-based and policy-based methods, is a viable solution. In fact, implementations like *FIRM* [48], *MIRAS*

[68], and *Harmony* [6] successfully integrated Actor-Critic methods in their research and showed beneficial performance in their respective problems. Although these studies employ decision-making Actor-Critic methods, they do not consider the effect of the agent's state space definition, which is the set of elements observed by the agent, with the selected metrics. As a result, there is currently no study that specifically examines how the inclusion of specific metrics in the definition of the state space affects the overall performance of the model. Nonetheless, the state space is a crucial component in the characterisation of the algorithm because, ultimately, it gives the agent the necessary information to take the right scaling decisions.

To summarise, even though there have been solutions that integrated multiple metrics to evaluate the autoscaling action, there is a lack of assessment about the interaction of such RL algorithms with a state space based on those metrics. Therefore, the aim of this degree project is to investigate the performance of model-free Actor-Critic methods, specifically Advantage Actor Critic (A2C), with respect to multiple performance metrics in the context of horizontal autoscaling in Kubernetes. These results could be used to make more informed decisions on how Actor-Critic algorithms behave in dynamic environments such as MSA and which metrics are suitable to evaluate the performance of a service.

1.2 Problem

The general problem that container autoscalers aim to solve is to keep the response time of a microservice running on containers in the range of what is defined in the SLA [38]. An example of SLA could be to serve 99.95% of requests below 200 ms. In this context, response time is the time elapsed from when the request is sent to a server to when a response is received back, known as Round Trip Time (RTT). As the number of input requests increases, the container receiving those requests may not be able to process all of them in a timely manner. Therefore, the role of the HPA is to scale out that service by creating multiple copies of it and thus distributing the load among these. In this way, the response time is brought to an optimal level again, which satisfies the SLA.

The most common parameter to determine the scaling behaviour of the HPA is the container Central Processing Unit (CPU) utilisation [25], [72]. In general, well before resources like CPU or memory are saturated, the scaling action is triggered. Moreover, these metrics are in fact very easy to measure and intuitively are related to how busy a container is in performing its tasks. However, the CPU utilisation may not be the ideal metric to measure every microservice performance because the relation between its utilisation and SLA may not be linear or, as in [69], in some cases high CPU utilisation still leads to acceptable performance.

Therefore, one alternative is to consider other metrics that can measure the application performance. However, it has been discussed in several studies that managing the trade-off between different performance metrics is in fact a challenge [71], as well as tuning the parameters during the autoscaling process for optimal resource provisioning [25]. Since these metrics have a fundamental role in understanding the health of a service, it is crucial to select the most appropriate one, or a suitable subset of them.

1.3 Research Question

We want to evaluate the impact of autonomous container orchestration in MSA with the A2C algorithm by comparing it to the baseline. As a baseline, we will consider the Kubernetes HPA scaling on the average CPU utilisation of the deployment. Therefore, we formulate the autoscaling task as an RL problem where the aim is to take at each autoscaling cycle the scaling decision that maximises a reward function.

The reward function is designed in a way that drastically penalises the violation of the given SLA, in the case in analysis the service latency, and penalises also overprovisioning of resources. Through experiments, we would like to analyse and conclude on the following question:

- ***RQ1:*** What is the impact of including specific container's performance metrics (e.g. CPU, memory, latency, requests per second (RPS)) in the state space on a model-free RL agent's ability to accurately predict optimal autoscaling actions?

And, in order to compare the RL model to the baseline, the following sub-questions:

- ***RQ2:*** How does the use of average CPU utilisation impact the QoS and the resource utilisation in the baseline autoscaler?
- ***RQ3:*** How does the model-free algorithm applied to the autoscaler perform compared to the baseline in terms of SLA assurance and resource utilisation?

Each of these questions will be answered in Chapter 5.

1.4 Goal

This degree project aims to study the performance of an A2C agent taking scaling decisions in a Kubernetes cluster where a demo MSA app is deployed. The work can be divided into the following sub-goals:

- Perform a comprehensive literature review aimed at understanding the state-of-the-art of container orchestration with machine learning, specifically with RL.
- Build a prototype autoscaler that is able to take scaling decisions based on the state of the environment.
- Understand the influence of the state space definition on the performance of the prototype autoscaler.
- Compare the proposed RL method with the baseline to solve the problem of optimal resource utilisation and SLA compliance in a Kubernetes cluster.

The main deliverable and results of the thesis are:

1. A viability evaluation of the model-free A2C approach to learn and perform the scaling action based on multiple container performance metrics.
2. A performance comparison between the proposed approach and the baseline.
3. An evaluation under different scenarios of the proposed solution.

Finally, we will also discuss the possible limitations of the proposed methods and suggest future improvements.

1.5 Purpose

The purpose of this thesis is to investigate the impact of the state space constituted by different sets of container performance metrics on the scaling accuracy of A2C models, which will be potentially beneficial for improving the overall scaling performance with RL methods. In the context of autonomous scaling decisions for microservices, these results could also help researchers to further evaluate and implement more advanced Actor-Critic algorithms that achieve better resource utilisation while being SLA compliant.

For example, when employing RL algorithms to replicate services, they could compare which performance metrics should be monitored and how the state space can affect the ability of the agent to take scaling actions in dynamic environments. In addition, the framework of this thesis could also be treated as part of a more advanced and sophisticated end-to-end solution. By including information about microservice dependencies in the system and predictive methods to overcome the inherently reactive nature of the implementation, it can be applied to real-world autoscaling problems. Finally, the methodology developed in this work and the obtained results could be generally helpful for researchers targeting similar scaling optimisation problems in container orchestration.

1.6 Methodology

Based on the autonomous scaling optimisation problem that we are investigating, the theory, methods, experiments, evaluation strategies and design decisions that we select, along with the tools and technologies employed to perform our measurements, are made towards answering the research questions. A detailed discussion of these decisions and our motivations is to be found in Chapter 3.

After designing and evaluating the simulation environment, we followed an empirical research methodology to perform accurate experiments concerning the behaviour of the A2C agent and the

baseline. Quantitative evaluations of both the agent's performance and the baseline are carried out considering performance metrics that reflect their ability to solve the problem, along with measurements on the effect of the state space on the A2C agent, with the goal to answer the research questions and objectives proposed in Sections 1.3 and 1.4. These evaluations are performed with the double purpose of helping the reader understand the data collected from the environment, and to compare to the selected baseline the results from our proposed method. Finally, an analytical discussion around the research questions based on the obtained results is performed.

1.7 Stakeholders

Spotify AB, the music streaming platform with which this research has been conducted, runs its services completely on the cloud and its infrastructure counts thousands of microservices running on containers, which are managed using GKE. The challenge they face with the container orchestrator tool is similar to any other company using that technology. It is in their interest to conduct a study that can highlight the benefits and limitations of autonomous solutions employing RL. The thesis will provide the company with a demonstration of the applicability of autonomous scaling with A2C methods on an emulation of a GKE cluster, and an extensive literature review that presents the state-of-the-art on the autoscaling problem, which could help in the selection of the most suitable approach for a possible future implementation.

1.8 Delimitations

This degree project is mainly concerned with studying in a simulated environment the performance of an RL autonomous HPA when different sets of metrics are considered to take scaling decisions. Furthermore, this approach is compared with the Kubernetes HPA baseline to assess the advantages and limitations of such a solution considering resource utilisation and SLA compliance.

This thesis does not address specific issues such as

training and convergence speed of the model, assessment of different DRL algorithms, scalability concerns in large-scale production environments, and integration issues in real-world scenarios. Due to the expensive cost and implementation overhead of this solution in real production environments, we limit our thesis to demonstrate the viability in a benchmark microservice application with synthetic traffic workloads.

In addition, due to the purpose of this degree project, we try to make assumptions that simplify the complexity of the real-world production environment. The limited number or traffic patterns considered, together with the limited number of service types and microservice topologies that could be measured, may in fact limit the evaluation of the model performance. Finally, we would like to remark that this thesis only represents an initial step toward a more holistic approach in autonomous scaling optimisation in Kubernetes clusters. A discussion on such limitations will be presented in Chapter 5.

1.9 Benefits, Ethics and Sustainability

The problem of container autoscaling optimisation is directly related to the sustainability of cloud computing. In fact, as the number of containers is scaled up, more computing power is needed to accommodate for such needs. Even though software containers are considered a virtualised environment, they nonetheless run on physical machines, which require energy and sophisticated cooling systems to function. At Spotify AB, all containers run in the public cloud Google Cloud Platform (GCP) which as of today is carbon-neutral and has the goal of becoming carbon-free by 2030 [56]. Therefore, computation run on their cloud should theoretically be carbon-free and sustainable. Moreover, it is known that machine learning models often require a lot of energy to function, therefore it has been our responsibility to limit the deployment and testing of these computationally-intensive resources to the extent we need for the project. Lastly, no personal and sensitive user data has been employed due to the nature of the problem, which only involves data about the performance of virtualised hardware components like containers.

To conclude, by optimising the functioning of the widely adopted

Kubernetes HPA, a reduction of the rate of carbon emissions from its operations is likely to happen.

1.10 Outline

The rest of this thesis is organised as follows. Chapter 2 introduces the necessary background knowledge to understand the project and related works. Chapter 3 presents the methodology applied in this context. Chapter 4 describes in detail the experiments that have been conducted and the following Chapter 5 discusses the results and evaluates the limitations, while Chapter 6 draws the conclusion of the work and indicates future directions.

Chapter 2

Background

This chapter provides basic background information about cloud computing, container orchestration and reinforcement learning. Additionally, this chapter describes MSA and service mesh as two important frameworks used for the experiments of this thesis. Finally, a discussion of related works is presented in the final section.

2.1 Cloud computing

Today, there is a high chance that more than one app installed in a smartphone or a web service like an e-commerce are employing cloud computing. In fact, cloud computing is a framework that allows companies with digital service offerings to rent computing power on-demand by only paying for the time they use it. Its main advantage is that companies do not have to buy, build and maintain the software and hardware equipment to host their services. Instead, it is the cloud provider that takes this responsibility and offers ready-to-use remote computers that can be accessed with an Internet connection [65]. This is a real game changer in the industry because, for example, a small startup that wants to ship a smartphone application to potentially millions of users does not have to build and maintain its own expensive data centre, but can have compute power ready on-demand and pay only for what is actually used. Moreover, those computing resources can be used for anything from data storage to data processing and streaming, to train machine learning models on specific hardware, to

simulate scientific experiments, play online videogames, or simply host an e-mail server or a web page.

Cloud computing has different deployment models and these include *public* cloud, *private* cloud or *hybrid* cloud. While public clouds are completely run by third-party cloud providers such as Google, Amazon, or Microsoft, private clouds are owned and maintained by the organisation that uses them. Private clouds still need to be run and maintained in private data centres by the companies adopting this model, nonetheless, they are built in such a way that leverage the benefits of cloud computing virtualisation. A combination of public cloud and private cloud is called hybrid cloud, and it is adopted by companies that rent some services from cloud providers while securing crucial parts of their business in their own private data centres.

On top of these deployment models, cloud computing offers three main service models, commonly known as *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)* and *Software as a Service (SaaS)*. Considering IaaS, the offer comprises infrastructural services like virtualised storage, networking, and compute power which can be managed with the highest control. In the case of GKE for example, we refer to PaaS because the underlying infrastructure is managed by the cloud provider, while clients use the hardware and software resources to develop what they need with Kubernetes, which is the platform of interest. Finally, SaaS are software applications that can be used by the end user with an internet connection and thus do not need to be installed locally. In this case, the whole application is fully managed and maintained by the provider.

To sum up, cloud computing is mainly used for infrastructure scaling, data storage, big data analytics, application development, and disaster recovery. It enables greater flexibility thanks to the on-demand availability of compute and storage resources through the Internet, which can be scaled in terms of number of machines and compute power when needed thanks to dedicated virtualisation technologies. Even though its cost can be consistent for very large clients, it is still a cost-effective solution compared to traditional infrastructure models where each company have to host and maintain their own data center and services.

Specifically to virtualisation, services hosted in the cloud typically

run in VMs, which are software-defined computers with their own operative systems and programs. One important advantage of VMs is that completely different services can run independently in the same host hardware. For example, in the same physical computer with Ubuntu 20.04LTE OS one could have a Virtual Machine (VM) with a macOS Sierra that would behave exactly as a Mac and one with a Windows 10 OS that could be used to download completely different programs. The fact of having a virtual computer makes it possible to host in a powerful hardware many other virtual computers, which can be created and deleted on-demand.

To conclude, because the cloud computing model offers such advantages, it is widely adopted by companies and startups, and it is in fact the underlying technology of many digital products and solutions available today.

2.1.1 Software containers

However, there is an additional layer of virtualisation that makes cloud computing even more flexible and helpful, which is called *containerisation*. Software containers are lightweight and portable software packages that contain everything needed to run an application, such as code, libraries, and settings [62]. They isolate applications from the underlying computer system, making it easy to deploy and run consistently across different environments. For example, one could create an app that runs in the Ubuntu VM, package it in a container file and copy that file in the Windows VM. In the latter, with a container runtime like Docker [63], the app can be run without modifications at the code.

The visualisation in Figure 2.1.1 will help the reader understand the transition from a traditional deployment, in which everything was residing in a single machine, to a containerised one, where every single component is virtually separated from the rest. While in the case of a virtual machine there is the redundancy of the operating system on top of the same hardware and all VMs are managed with a *hypervisor*, in the case of containers there is a single OS and typically a single app for each container. In the same figure, two important components are the *hypervisor* for virtualised deployment and the *container runtime* for the container case.

Examples of hypervisors available in the market are VMWare ESXi and VirtualBox, and their role is essentially to monitor VMs and abstract hardware resources of the host machine, like CPU, memory and network interfaces, to translate virtual hardware requests into requests for physical resources [1]. On the other hand, container runtimes like *containerd* [12] make use of kernel features of the host OS, like *namespaces* and *cgroups*, to abstract and isolate applications into containers [10]. This provides significant performance benefits compared to hypervisors because instructions from containers are executed in the host hardware directly and do not need to be translated.

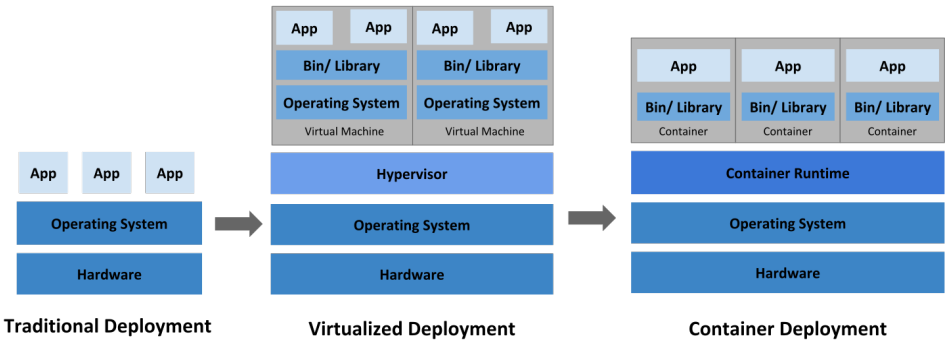


Figure 2.1.1: The evolution of deployments from traditional to container.
Source: `https://kubernetes.io/docs/concepts/overview/#going-back-in-time`

Docker is a very popular container engine that uses containerd to manage the lifecycle of containers. It is built with a client-server architecture in which the developer interacts with the Docker runtime to operate with containers [16]. A typical container's lifecycle includes five states:

1. *created*, when a container is created from a Docker image
2. *running*, when the container executes the commands mentioned in the image
3. *paused* and *unpaused*, when the command running in the container is paused or unpaused, respectively
4. *stopped*, when the container main process is gracefully stopped

from executing and the container is shut down

5. *deleted*, when the container is removed

A Docker *image* is essentially a template that contains all files needed to run an application inside a container. This image is used to create a container which can be run with the instructions that are defined in the template. For example, a Docker image for a web server has all the required packages and instructions to serve Hyper Text Transfer Protocol (HTTP) requests coming from external clients. And, once the image is built and run in a container, that container will be ready to process all incoming traffic.

Similarly to the image for the web server, there are thousands of other different images that are publicly available, ready to use, and highly customisable, which makes Docker a very popular container engine, counting up to 11 billion monthly image downloads [15]. Finally, the fact that containers can be created and deleted considerably faster compared to VMs because there is no need to boot the OS, makes them the ideal technology to satisfy demands for flexibility and scalability in cloud platforms [53].

2.1.2 Microservice architecture

Container technology would not have claimed all its popularity if a fundamental architectural change in the way in which software applications were build had not occurred. *Microservices architecture* is a software architecture style which have gained a lot of popularity in the late years in the industry because of its advantages compared to traditional monolithic architectures [2]. Essentially, MSA is a framework that emphasizes the division of the system into smaller, lightweight and decoupled services that together are built to perform a cohesive business function [4]. Therefore, a *microservice* can be defined as the minimal independent component of an application, and containers are the technology that enables the packaging of such small components into functional entities that can be easily managed.

In the past, a traditional application would have been built with the so-called *monolithic* architecture, meaning that all its components are thightly coupled together in a single, interconnected unit. On the opposite, the microservice architecture is a modular approach

that decomposes the application into smaller, independent services that can thus be developed, deployed and scaled independently. For example, a monolithic architecture for a website used to order some items would include in the same codebase the user authentication, the database access and the functions to process the order. While, with a microservice architecture, the same service might be decoupled in a service handling user authentication, another reading and writing in the database, and one to process the order. This latter approach increases agility, maintainability, separation of concerns, and scalability because every component can now be treated independently. As mentioned above, containers enhance the use and maintainability of such components because they can host an individual microservice with all the executable files, dependencies and libraries needed for it to work properly [53].

However, decoupling a complex application into smaller independent sets introduces new challenges [4]. Since these services are decoupled, their communication over the network is more involved and service discovery can be a real issue. Moreover, there can be concerns over security management, data sharing and overall performance. However, when these problems are addressed appropriately, the benefits of the architecture can overcome its disadvantages.

2.1.2.1 Example of microservice architecture

To picture more concretely what a microservice architecture appears like in a real deployment scenario, in the following we report the demo application referenced in this thesis. The code of the underlying infrastructure, together with the installation instructions are all publicly available [8].

The microservice application is called *Online Boutique* and consists of 11 microservices for an e-commerce website where users can browse items, add them to the cart, and purchase them. Moreover, it has been used by Google to demonstrate the use of microservice technologies and by researchers as benchmark to study MSA and conduct experiments [69]. In particular, the experimental setup developed for this thesis was independently adapted from `v0.5.2` of the original *Online Boutique* project, publicly available as GitHub repository [20].

The diagram in Figure 2.1.2 represents the interconnections between the 11 different services, represented with a rectangular box, in terms of service calls. Note that the user on top only interacts with *frontend*, which is the website landing page presenting the clickable interface. Depending on the action performed by the user, different sets of services might be called. For example, a typical user journey may look like:

1. landing on the e-commerce website
2. browsing for some products
3. adding one product to the cart
4. proceeding to checkout

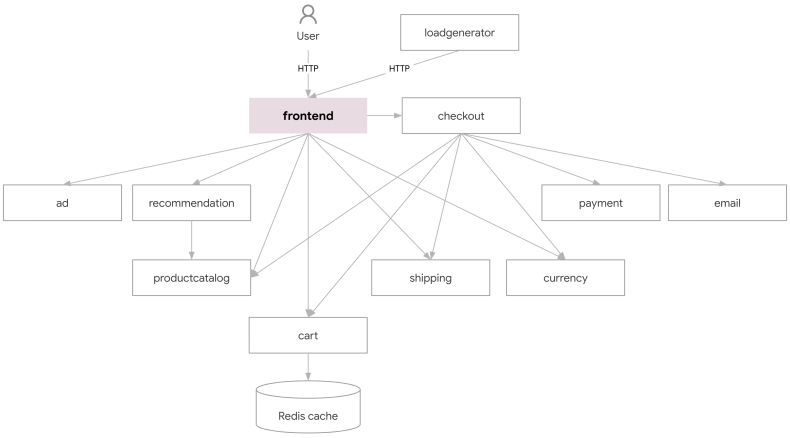


Figure 2.1.2: *Online Boutique*'s microservice architecture. Source: `https://github.com/GoogleCloudPlatform/microservices-demo/blob/main/docs/img/architecture-diagram.png`

For each of the above steps, a sequence of microservices will be called to perform the required functions:

1. The user sends an HTTP request to *frontend* to visit the website. In order to display the main page, *frontend* will call *ad* to display advertisements, *productcatalog* to display the available products, *recommendation* to perform a product recommendation based on user's interest, which will in turn independently call *productcatalog* for checking which products to recommend. All this information will be returned in the landing page and thus to the user.

2. The user clicks on some product to see its picture, price, and other details. At this stage, *frontend* will forward the user's request to *productcatalog*, which will respond with the details of the desired product.
3. The user adds the desired product to the cart. Here, *frontend* will forward the request to *cart*, which will add the desired `productID` and `userID` to the cart and save the change in the database *Redis cache*.
4. Finally, the user buys the product from the cart. At this point *frontend* would call *checkout* which calls six other services to place the order. *Checkout* calls *cart* to retrieve from *Redis cache* the user's cart items. Also *productcatalog* will be called to request the product's details based on the `productID` returned by the *cart*. Then, the payment transaction will be performed by *paymentservice* based on the currency selected by the user and the rate of conversion computed by *currency*. If the transaction is successful, *checkout* will call the *shipping* microservice to initiate the shipping process and an order confirmation will be sent to the user via e-mail with the *email* microservice.

From this rather simple example some important considerations concerning MSA can be drawn. In this architecture, different components communicate with each other via standard Internet protocols and this can generate a lot of internal networking traffic. Such protocols typically include HTTP and Hyper Text Transfer Protocol Secure (HTTPS), Transmission Control Protocol / Internet Protocol (TCP/IP), and Google Remote Procedure Call (gRPC). Moreover, not all services are needed at the same time and depending on the user's behaviour, some services may receive fewer requests. In addition, services are identified based on their business function and can be developed and maintained independently, for example, the *ad* microservice could be upgraded without touching the code of *email* service and, conversely, if the *email* service breaks, this does not prevent *ad* to function properly or the entire website to shut down. This independence between microservices improves also the flexibility when developing them because different programming languages can be used for different components. For example, *frontend* is written in Go while *checkout* in C# because these languages suit best the specific requirements of those microservices.

In the *Online Boutique* implementation, every microservice is defined as a Docker image and thus deployed as a container in a host machine. The image defines the behaviour of the microservice, in particular it contains the code, defines its specific dependencies, and provides information on how to call other microservices in the application. In this way, every container will be responsible for each microservice and the container runtime will manage the lifecycle of the whole application in the dedicated hardware. Ultimately, this ability to maintain the separation between microservices in the same host system thanks, to *namespaces* and *cgroups*, makes containers the most suitable technology for this architectural style.

2.2 Container orchestration

However, the simplified case presented above, in which only one user visits the website, is unrealistic. For this reason, *loadgenerator* is used in *Online Boutique* to simulate the behaviour of multiple users operating on the website at the same time. In fact, when a lot of traffic is generated, the application is stressed from the incoming load and performance bottlenecks become a concern. Some *backend* services, which is a general term to indicate those services that are called by a *frontend*, need to be scaled in order to satisfy the users' requests in time. Additionally, in more realistic real-world scenarios some applications in the cloud can count thousands of microservices [17], that when replicated can reach a considerable number of containers, and some services may receive millions of requests per second, making it necessary to have tools that automatically manage these systems.

Container orchestrators are software programs that automate the management of containers at scale, allowing for efficient deployment and scaling of applications. They provide a centralised platform to manage containerised services across multiple host machines. Most importantly, they enable scaling by dynamically adjusting the number of container instances based on demand, ensuring optimal resource utilisation and high availability of applications.

2.2.1 Introduction to Kubernetes

The de-facto standard orchestrator is Kubernetes (K8s) [61], an open-source project initially developed by Google and later open-sourced and donated to the Cloud Native Computing Foundation (CNCF). Some of the most useful features that Kubernetes offers are high availability and no downtime of containers, scalability and high performance, and disaster recovery [40]. Kubernetes is considered a PaaS and it can be deployed in physical machines, VMs and cloud environments, like in GCP for example.

The main features that Kubernetes brings are service discovery and load balancing, storage orchestration, automated rollouts and rollbacks, automatic bin packing, self-healing, and secret and configuration management [40]. With *service discovery* it is meant the ability of K8s to make containers discoverable in a network by using a Domain Name System (DNS) server, thus allowing them to communicate with each other, while *load balancing* refers to distributing the load among different containers. *Storage orchestration* is the possibility to mount different storage systems in the containers, such local storage or storage provided by cloud providers for example. An *automated rollout* is a non-disruptive way of updating components in the cluster, for example upgrading a container image to a newer version, and can be initiated with a single instruction that is automatically replicated to all targets. In case issues are encountered during a rollout, Kubernetes also provides *automated rollback* functionality to revert to the previously known stable version. Another key feature of the orchestrator is *automatic bin packing*, which refers to the ability to schedule containers based on the availability of resources like virtual CPU (vCPU) or virtual Memory (vMemory) in the nodes of the cluster. In addition, K8s manages the lifecycle of its containers with a *self-healing* mechanism which restarts containers that fail, checks their health and routes traffic to them only once they are deemed ready. Finally, K8s allows the storage and management of sensitive information such as passwords, Secure Shell (SSH) keys, or authentication tokens that can be used to restrict access to certain resources to the users of the platform or that can be used by containers to authenticate themselves to secure services.

2.2.2 Kubernetes architecture and components

The purpose of this section is to provide an overview of the architecture of Kubernetes, the role of its components, and how they interact with each other and the containers.

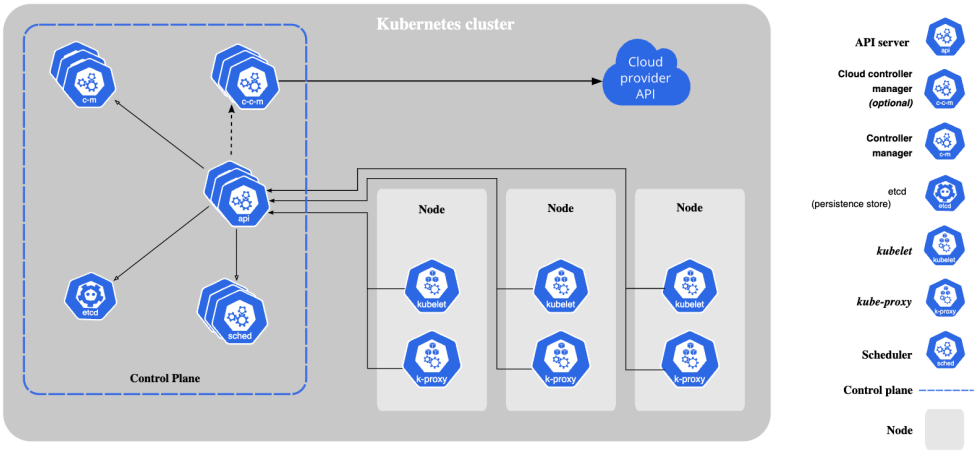


Figure 2.2.1: Kubernete's architecture. Source: `https://kubernetes.io/docs/concepts/overview/components/`

Figure 2.2.1 represents the architecture of a typical K8s installation. The group of machines that host all the containers managed by K8s is called *cluster*, depicted as the background grey box in the picture, while each individual machine hosting containers is referred to as *node* [29]. These components can be divided into *Control Plane components*, responsible for global decisions about the cluster, *Node components*, dedicated to the lifecycle of containers, and *pods*, which are the smallest deployable computing unit of a Kubernetes cluster.

2.2.2.1 Control Plane components

These components are usually deployed in the *master* node of the cluster, which is that machine that controls all other nodes, called *worker* nodes. In fact, the Control Plane acts as the orchestrator of the Kubernetes system, and, for example, it is responsible for starting up pods and scheduling them into the worker nodes, and responding to cluster events and instructions given by the users.

There are four main components in the Control Plane, plus one optional component that is activated only when Kubernetes is deployed in the

cloud.

- *kube-apiserver*, it is the main communication endpoint of the Control Plane and it is the intermediary between all components in K8s. Specifically, it is an Application Programming Interface (API) server which defines how the application should respond to the requests that it receives from all other units. Moreover, it provides up-to-date information on the cluster's state and on all its parts [26].
- *etcd*, it is a distributed and consistent key-value database that stores all cluster's data [19], for example, information about each node's state, how many pods it contains, resources consumed, Internet Protocol (IP) addresses and everything else needed by the *kube-apiserver* to know the state of the system.
- *kube-scheduler*, it is the component that takes scheduling decisions. In particular, when a new pod is created, it decides in which node to place it based on a collection of data such as resource requirements, scheduling policies and other constraints.
- *kube-controller-manager*, it runs the controller processes, which are basically a series of non-terminating loop cycles that periodically check the state of the cluster and make or request changes where needed [13]. Examples of controllers are *Node* controller, *Job* controller, *EndpointSlice* controller, *ServiceAccount* controller.
- *cloud-controller-manager* (optional), it is fundamentally a *kube-controller-manager* that is deployed when K8s runs within a cluster in the public cloud. Basically, it provides a way to use the control loop over provider-specific items.

2.2.2.2 Node components

As reported above, the Control Plane is used to manage worker nodes, and each of them have three main components: *kubelet*, *kube-proxy*, and the *container runtime*. The role of the node is to host containers and provide the hardware resources and the OS features for containerised applications to run properly.

- *kubelet*, it is the main agent that runs in a node and makes sure that each hosted pod is running healthy according to their individual specifications provided under *PodSpecs* by periodically running diagnostics on them [28].
- *kube-proxy*, it is a network proxy that manages the networking traffic between pods, which includes Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Stream Control Transmission Protocol (SCTP) streams [27]. It implements the *Service* concept in Kubernetes, as explained in Section 2.2.4.
- *container runtime*, it is the software responsible for running containers, and it provides the same virtualisation features presented in Section 2.1.1.

2.2.2.3 Pods

A *pod* is the smallest unit of computing that can be deployed in a Kubernetes cluster [43]. Essentially, it is a group of one or more containers that are tightly coupled to form a cohesive service, and it provides a logical abstraction of them in such a way that they can be managed by K8s and software developers as a single entity. Therefore, a user usually interacts with a pod and not directly with the container or containers inside it. In addition, because pods host containers that are normally coupled together and have to share storage or network resources, these are always co-located and co-scheduled to run in the same node.

For example, in the case of a container for a web server that serves files from a shared storage, called a *volume*, there could be a *sidecar container* that periodically synchronises files from a remote source and updates the files in the volume so that the web server can display the latest version of those. In this case, the pod is designed to group together the web server and its sidecar and connect them to the same shared volume. The *kube-scheduler* will schedule this pod in a single node so that both containers will reside together.

Moreover, pods are to be considered relatively *ephemeral* in the sense that they are disposable entities that can be created, destroyed and recreated dynamically. For example, when a pod finishes its execution

it is deleted from the node to free the resources it was requesting. Moreover, the ephemeral nature of pods align with the principles of scalability and high performance of Kubernetes.

Similarly to a container lifecycle, a pod lifecycle has different phases. These include *pending*, while the pod is waiting to be scheduled to a node, *running*, when at least one container has started, *succeeded* when all containers inside the pod are healthy, or *failed* if at least one container has terminated in failure [42].

To sum up, Kubernetes is used to orchestrate containerised applications which, for simplicity and tractability, are abstracted as pods. Fundamentally, K8s manages these entities, which are scheduled and replicated inside nodes to provide them with the necessary hardware resources such as storage, networking, memory and CPU. Pods are scheduled into nodes based on the resources they request and on the availability of those worker machines, which can be physical machines or VMs. Therefore, a deployment inside a K8s node will resemble a container deployment as the one depicted in Figure 2.1.1. Note that components such as *etcd*, *kube-apiserver*, *kube-proxy*, etcetera, are themselves deployed as pods. Finally, a collection of worker nodes, together with the master node, is called a K8s cluster and these components basically make the underlying architecture of Kubernetes.

2.2.3 Addons

In addition to those reported above, it is possible to extend the functionality of K8s with more components, called *addons*. These provide cluster-level features and include third-party implementations for networking and network policy, service discovery, visualisation and control, infrastructure, and monitoring [23].

With respect to container resource monitoring, two available approaches are *resource metrics pipelines* and *full metrics pipelines* [60]. The first pipeline involves the *kube-metrics-server* [30], a lightweight in-memory database, that only collects a limited set of metrics related to nodes and pods, such as CPU and memory usage. It collects those values from the *kubelet* in each node and gains access to the resource consumptions of each individual container in the node. These metrics are then published to the Kubernetes API at

`/metrics/resource/v1beta1`, and this endpoint is used by the HPA and other utilities to retrieve the state of the cluster. However, to get a richer set of metrics, the *full metrics pipeline* may be used. In this scenario, a third-party software can be used to *scrape* a variety of different metrics from the components in the cluster, which may include application-specific metrics as well as container-level metrics, and exposes them to Kubernetes.

In the context of this project, two such tools used to collect metrics are Prometheus, which will be presented more thoroughly in Section 2.4.1, and the *kube-state-metrics* server [39], and the metrics provided by these services will be presented in Section 3.4.

2.2.4 Other Kubernetes concepts

Besides to the K8s architecture, it is necessary to introduce some other fundamental concepts in Kubernetes. Among others, the most important to understand in the scope of this thesis are three, namely *namespaces*, *deployments* and *services*.

2.2.4.1 Namespaces

Namespaces offer a way to isolate groups of different resources within the same Kubernetes cluster [35]. It is not strictly necessary to use them, but they are a useful feature when a cluster need to be logically divided for multiple users. For example, all objects created by the Kubernetes system resides in the `kube-system` namespace. Another example could be to deploy an application with different microservices in a namespace called `app`, and other applications for monitoring purposes in a namespace called `monitoring`. This division allows higher control over the pods. In the scope of this thesis for example, this logical division becomes useful when we want to run multiple experiments in parallel on the same cluster. In fact, the same application can be run in two different namespaces at the same time, allowing to test different things with the same setup.

2.2.4.2 Deployments

Usually, pods are not created directly by manually calling the Kubernetes API, instead, they are created using workload resources

such as Deployments or Jobs [43]. In essence, deployments are templates used to provide declarative updates to pods [14]. For example, when deploying an application, one often creates a *deployment file* for each microservice, where specific configurations are declared. For example, in this file one may declare the amount of CPU and memory they expect the pod to use, which container images to use, the namespaces in which to deploy the pod, among many more others. An example of deployment file may look like the following.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: productcatalogservice
spec:
  template:
    metadata:
      labels:
        app: productcatalogservice
    spec:
      containers:
      - name: server
        image: productcatalogservice:0cff6aa
        ports:
        - containerPort: 3550
        resources:
          requests:
            cpu: 250m
            memory: 64Mi
          limits:
            cpu: 300m
            memory: 128Mi
```

2.2.4.3 Services

To expose services in the network and enable communication between pods, Kubernetes uses Services [52]. In a highly dynamic environment like Kubernetes, where pods are created and destroyed continuously, it is extremely important to keep track of IP addresses. In fact, as these addresses indicate where in the network to find a pod, if the pod

is destroyed, that information is lost with it. Moreover, if there are multiple replicas of the same microservice, a client connecting to it does not really care to which replica it should connect, as long as its request is satisfied by any of the available pods, therefore, it should not keep track of their IP addresses. Services in Kubernetes are used to cope with these issues, as they abstract and decouple the pods from the network by managing the IP addresses of replicas behind that *Service**. In this way, clients can connect to *Services*, and these will take care of how to distribute the incoming requests to the servers in the pods.

2.2.5 Horizontal Pod Autoscaler (HPA)

One of the most useful features of Kubernetes is perhaps its Horizontal Pod Autoscaler (HPA), which is an automatic scaling mechanism for pods that has the aim of adjusting replicas to the varying demand for their services [22]. Specifically, to scale *horizontally* means to increase the number of pods, as opposed to *vertical* scaling, which refers to the assignment of more resources like CPU and memory to a single pod. The number of pods can be *scaled-out*, when the number is increased, or *scaled-in*, when the number is reduced. Thus, the principal role of the HPA is to scale-out or scale-in pod replicas according to the dynamic load of requests they receive. For example, when a microservice is overloaded with requests from a client, its performance may degrade because it is not able to respond to all of them in a timely manner. For this reason, another copy of the same microservice can be created, and requests be distributed among the two replicas. In this way, each replica will have fewer requests to satisfy individually, and they can both respond according to the normal performance.

This feature of K8s is extremely useful because it allows companies to maintain the optimal quality of their services, independently of the volume of the demand. However, there are two main concerns that always arise when using an autoscaler. The first regards the QoS and SLA assurance for the services, which may degrade when there is *underprovisioning*. Essentially, it occurs when resources are provisioned below the optimal level, resulting in pods not being

*Please note that in the following we will refer to Kubernetes' *Service/s* with upper-case *S* as the networking concept, while *service/s* with lower-case *s* to indicate microservices or the business function they offer.

able to perform at their best and possibly violating agreements, for example of an SLA of satisfying 99.95% of requests below $200ms$ could not be met. The second is related to the monetary cost of having many under-utilised pods, thus resulting in a waste of resources, ending up in a situation called *overprovisioning*. In this case, because cloud users pay for the resources they actually provision and use [65], it is also important not to reserve more resources than needed. Ideally, autoscaling helps in avoiding these two scenarios by managing automatically the scaling. The strategy that is used by the HPA is based on performance metric thresholds that are known to maintain the desired QoS, and, if exceeded, trigger the scaling action. In fact, the algorithm that governs the HPA is based on a control loop that periodically checks the state of a deployment and computes the desired number of replicas as in Equation 2.1.

$$N' = \left\lceil N \times \frac{\zeta}{\eta} \right\rceil \tag{2.1}$$

The desired number N' is the ceiling function of the current number of replicas N multiplied by the ratio between the current metric value ζ and the desired metric value η. As an example, consider the case in which the desired metric value is *average CPU utilisation* of the deployment set to 70%, and that currently there are two active replicas. If the deployment is overloaded, for example with the CPU utilisation across the two pods being 120%, the HPA would compute the desired number of replicas as

$$N' = \left\lceil 2 \times \frac{120}{70} \right\rceil = 4 \tag{2.2}$$

resulting in a scaling-out action. Now, if the utilisation drops to 15% because of a decrease in workload, with 4 current replicas the number of desired replicas will drop to 1 for the same calculation, resulting in a scale-in behaviour.

By default, the HPA executes the control loop once every 15 seconds on each targeted pod. If a target resource utilisation is specified, for example 70% CPU, the controller computes the utilisation value as the percentage of the resource requested by each container in the pod. Then, it returns the mean value across all targeted pods. Alternatively, when a raw metric value is set, for example $500m$ cores of CPU, the raw

value obtained from the Metrics Server API is used directly to compute the mean value across pods. Lastly, if multiple metrics are specified, the calculations reported above are done for each metric, and then the largest of the desired replica counts is chosen to scale.

In order to avoid many scaling actions following rapidly fluctuating metrics, the HPA is equipped with a stabilisation window flag called `stabilizationWindowSeconds`, which can be set to any value, for example, 300 seconds. In fact, before taking the scaling action, the controller considers all HPA scaling proposals in the past time duration corresponding to the stabilisation window, and executes only the highest proposed value. By default, the HPA controls the scale-down behavior in a time window of 5 minutes.

An example of Kubernetes manifest that defines an HPA is the following. Here, the target of the HPA named *product-hpa* is the deployment called *productcatalogservice*, which should maintain the number of replicas between 1 and 15, and an average CPU utilisation of 70%.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: product-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: productcatalogservice
  minReplicas: 1
  maxReplicas: 15
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

However, the HPA comes with its own limitations. Two such limitations are the reactive nature of the autoscaler, which shows the

worst performance when there are sudden changes in the workload, and the reliance on infrastructure metrics only, mainly CPU load, to measure the performance of a pod. For what concerns the first limitation, the most common approach is to overprovision by lowering the threshold in such a way that the system is never saturated, or, a more sophisticated solution, is to deploy an machine learning (ML) model that is able to predict the incoming workload peak and scale up the replicas in advance [21].

The second limitation, which is in fact the subject of this degree project, has to do with how the microservice performance is measured. Currently, the HPA, as well as the Vertical Pod Autoscaler (VPA), are dependent on a manual procedure involving setting up thresholds such as CPU utilisation, the minimum and maximum numbers of replicas for each deployment [61]. Even though these are the default and most commonly used metrics in Kubernetes autoscaling, others candiadtes are for example Random Access Memory (RAM), network I/O, disk, response time, number of requests, and custom application metrics [61]. Other metrics concerning the QoS, often used as Service Level Indicator (SLI), are latency, error rate, throughput, availability, queue length, success rate, response size, time to first byte, etc. The main concern with choosing which metric to adopt, or a combination of metrics, is that it has a direct impact on the accuracy of the autoscaling decision. Therefore, a wrong metric could lead to over/under-provisioning, involve cold initiation of containers, cause delays in resource allocation, and a general degradation of the service [25]. This is the reason why research has been conducted on measuring the autoscaler performance with metrics other than CPU.

2.3 Service Mesh

Despite the benefits that the microservice architecture brings in terms of efficiency and productivity in the development and deployment phases of an application, the complexity of its operation during runtime is a challenge [31]. In particular, traffic control and other communication related concerns must be addressed properly in cloud-native applications, which are those designed specifically to run and exploit the benefits of cloud environments [31]. A promising approach to face this situation is *Service Mesh*, a fully-manageable

infrastructure layer framework for service-to-service communication designed to standardise the runtime operation of applications in the cloud [31].

The fundamental features of service mesh technology are [31]:

1. Service discovery: since in MSA the state of the microservices and their location change dynamically over time, service mesh must be able to keep track of the services in the network to make them discoverable;
2. Load balancing: the load balancer distributes traffic among services and replicas according to latency and state of a microservice;
3. Fault tolerance: because the network layer is considered unreliable by design, service mesh is equipped with failure-handling capabilites by directing traffic only to healthy services;
4. Traffic monitoring: the ability to capture all the service-to-service traffic and report it with metrics such as latency, traffic volume, error rates, etc.
5. Circuit breaking: it is the ability to recognise an overloaded service and avoid directing further requests to it, which would possibly bring the service to complete failure and further damage the QoS;
6. Authentication and access control: the possibility to enforce policies on the application to define which services are allowed to communicate with other components, and prevent unauthorised requests to access a service.

There are multiple products and projects in the cloud industry that implement service mesh, such as Istio [58], Linkerd [41], AWS App Mesh [64], Anthos Service Mesh [3], among others. In the context of this thesis, Istio service mesh is implemented to monitor the microservice application and provide load balancing.

2.3.1 Istio Service Mesh

Istio, as any other service mesh infrastructure, is logically decomposed in a *data plane*, responsible for the communication between services,

and a *control plane*, the brain of a service mesh, responsible for traffic routing configurations [58]. A basic representation of its architecture is depicted in Figure 2.3.1.

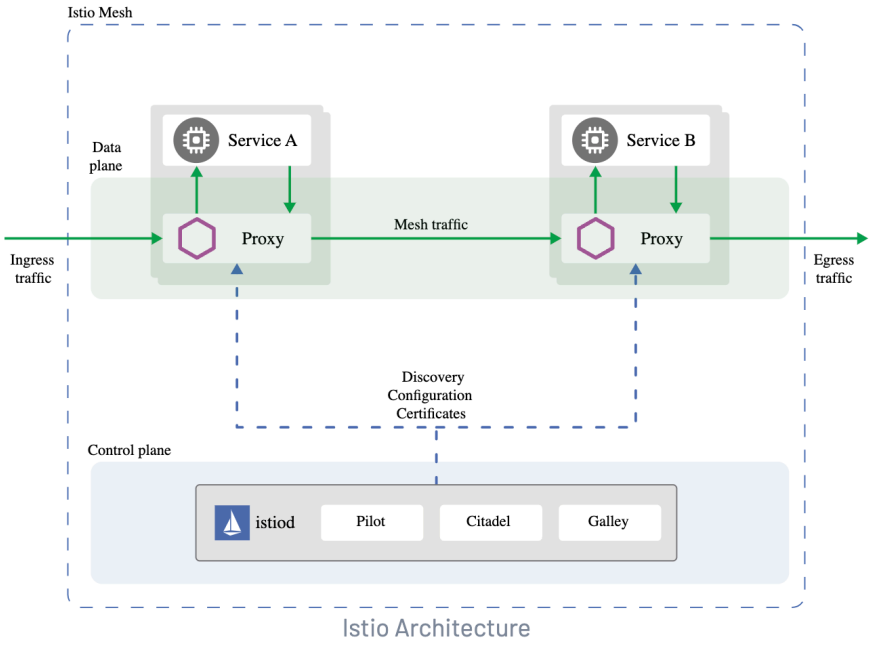


Figure 2.3.1: Istio Service Mesh architecture. Source: `https://istio.io/latest/docs/ops/deployment/architecture/`

2.3.1.1 Control Plane

The Istio daemon *istiod* constitutes the control plane, and it is deployed itself as a pod which incorporates functionalities of service discovery (Pilot), configuration (Galley), and certificate generation (Citadel) [7]. By default, in a Kubernetes cluster it is deployed under the namespace `istio-system` and the user interacts with it through the Istio API server with the command `istioctl`. Therefore, it receives configuration data from users or other management components, validates and processes it, and distributes the configuration to Envoy proxies [58]. It injects sidecar proxies when a pod is created, collects telemetry and monitoring data from them, and acts as the Certificate Authority that validates all security certificates within the mesh.

2.3.1.2 Data plane

The data plane is composed of a set of Envoy proxies [18] deployed as sidecar containers on a per-pod basis. These high-performance proxies mediate all incoming and outcoming traffic from the service mesh, meaning that services communicate with each other only through proxies, which handle all routing and protocol security features. Most importantly, it requires no code changes in the server application to enable an Envoy proxy and exploit Istio features such as monitoring and networking.

Specifically, Envoy is a cloud-native project developed by Lyft which abstracts the network from the microservice by providing common features like HTTP and gRPC support, load balancing, observability and API configuration management in a platform-agnostic manner [18]. Within the service mesh model, Envoy acts as a gateway to the network, where requests arrive via either ingress listeners, taking requests from external nodes to the local application via Envoy, or egress listeners, taking local requests and directing them to external services via Envoy [32]. Moreover, Envoy can be used in different configurations within the mesh, such as internal load balancer, ingress/egress proxy on the network edge, or in multi-tier topologies to improve scalability and reliability [32].

2.4 Cloud monitoring tools

In this section, we introduce two monitoring tools that are used in combination with Istio and Envoy proxies in this research, and which are widely adopted in the cloud industry as well. Prometheus [47] is a time-series database used to store pod metrics, while Grafana [24] is a software used to query and visualise timeseries data.

2.4.1 Prometheus

As mentioned above, Prometheus is an open source project to store metrics as time series data, meaning that each sample is stored with the time at which it was recorded. It can be deployed as a standalone pod in a Kubernetes cluster to monitor its components by scraping metrics from multiple sources. Its main features include a

multi-dimensional data model for time series data, a dedicated query language called *PromQL*, multiple code libraries available to interact with the Prometheus API, autonomous server nodes, pull model to scrape metrics over HTTP connections, the ability to push time series to external gateways for visualisation, built-in dashboards for simple graphing [47].

2.4.1.1 Data model

Time series are uniquely identified by their *metric name* and optional key-value pairs called *labels* [44]. The name specifies the metric that is measured, for example `container_cpu_usage_seconds_total`, and the label, or a combination of them, identifying a specific dimension of that metric, for example `container_cpu_usage_seconds_total{namespace="default", pod="frontend"}` identifies the metric relative to the pod called *frontend* in the *default* namespace. These metrics are sampled with custom millisecond precision timestamps and return `float64` values, i.e. numbers with floating point digits encoded in 64 bits.

2.4.1.2 Metrics types

Metrics can be of four main types: counter, gauge, histogram, and summary [46].

- Counters are cumulative metrics that are used to represent monotonically increasing values. For example, a counter to represent the total number of requests served by an instance. These values are reset to zero once the instance fails and it is restarted.
- Gauges are metrics whose value can increase or decrease in time. For example, the CPU utilisation of a pod that dynamically changes.
- Histograms are metrics that aggregate values by samples in configurable buckets. As an example, a metric may count the number of requests falling into each interval bucket of $10ms$ in the time interval from $0s$ to $10s$.
- Summaries are similar to histograms but different in the sense that they calculate configurable quantiles over a sliding time

window.

The specific query language for Prometheus, called PromQL, gives the possibility to use in its query arithmetic, trigonometric, logic comparison, and set binary operators, among many other vector operators, in addition to functions that allow to manipulate metrics, such as `sum()`, `rate()`, `count()`, `histogram_quantile()`, `avg()`, and `label_replace()`.

2.4.1.3 Jobs and instances

In Prometheus, an *instance* is the scraped endpoint, which usually corresponds to a single process. A *job* is a collection of instances with the same purpose, for example a process replicated for scalability or reliability [45].

Two examples of jobs which are used in this thesis are `kubernetes-nodes-cadvisor` and `kube-state-metrics`. The former is used to scrape resource metrics from Kubernetes nodes from the Container Advisor (cAdvisor), such as CPU and memory, which are those whose names begin with `container_`. The latter is used to generate metrics about the state of Kubernetes objects, such as the metric named `kube_pod_info` to count the number of available replicas in a deployment.

2.4.2 Grafana

Grafana can be used in combination with Prometheus to enable a complete monitoring stack in a microservice architecture [24]. In fact, it is an open source software that can be deployed in a Kubernetes cluster with its official Docker image that provides the user with a web interface where timeseries data can be visualised with dashboards.

Essentially, Grafana allows to build custom dashboards, and query endpoints such as Prometheus with PromQL, to immediately visualise data on the dashboard. Additionally, timeseries data can be further manipulated with filters and eventually downloaded locally in a machine.

2.5 Reinforcement Learning

The automation of orchestration processes for complex workloads in large-scale cloud computing systems remains uncertain due to the increasing diversity and dynamism of such environments [71]. To tackle this challenge, ML algorithms are employed in container orchestration systems for performance analysis and prediction, and for resource provisioning. In particular, decision-making models such as RL algorithms have been employed in resource provisioning to autonomously operate on scaling policies [71]. In fact, RL solutions are characterised by a generality and flexibility that only requires the specification of the desired deployment objectives, without explicitly programming how they should be obtained [49]. This provides a tremendous advantage in complex dynamic systems because the RL model can learn the scaling policy automatically without prior expert knowledge of the system. In the following, we will introduce those high-level concepts needed to understand the key aspects of RL related to this thesis.

2.5.1 Finite Markov Decision Processes

The general problem we aim to solve is to have an algorithm that continuously takes the right scaling action in the right circumstances. In this respect, it is possible to formalise such sequential decision-making problem using a mathematical framework known as finite Markov Decision Process (MDP) [57]. In this framework, there is a decision maker called *agent*, which interacts with an external *environment* by taking certain *actions* which alter its *state*. The goal of the agent is to maximise the reward provided by the environment as the outcome of its actions, and learn which behaviour leads to the maximisation.

In a finite MDP, the set of states $\boldsymbol{S}$, the set of actions $\boldsymbol{A}$, and the set of rewards $\boldsymbol{R}$, have a finite number of elements. At each timestep $t = 0, 1, 2, ...$, the agent observes the state of the environment $S_t \in \boldsymbol{S}$, and, based on this state, takes an action $A_t \in \boldsymbol{A}$. At the next timestep $t + 1$, the environment transitions to state $S_{t+1} \in \boldsymbol{S}$ and receives the numerical reward $R_{t+1} \in \boldsymbol{R} \subset \mathbb{R}$, which is relative to the action A_t taken at state S_t. The idea is illustrated in Figure 2.5.1, where the *sequence of decisions* starting from $t = 0$ can be represented as

$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$ Therefore, given the state-action pair (S_t, A_t), the *reward function* can be thought as a function that maps actions and states to rewards, $f(S_t, A_t) = R_{t+1}$.

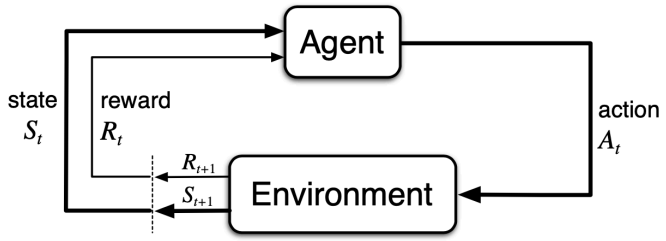


Figure 2.5.1: The agent–environment interaction in a MDP. Source: [57]

Moreover, because in sequential decision-making problems we are interested in characterising the environment's dynamics in such a way that the agent learns the best action for any given state, from a theoretical perspective a finite MDP becomes extremely useful because it does just that. In fact, since the sets of actions, states, and rewards are finite, the agent can learn the probability distribution by observing the frequencies of transitions and rewards occurring in the environment during its interactions, known as *transition probabilities*. By collecting data and keeping track of the observed state-action pairs and the resulting next states and rewards, the agent can estimate the probabilities associated with each state-action pair. This learning process enables the agent to build an approximation of the true underlying probability distribution and use it to make informed decisions. Therefore, the dynamics in a finite MDP can be completely described by the function

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

where, given the state-action pair (s, a), we can know the probability of ending up in state s' with consequent reward r [57].

To conclude, the power of MDP is that any problem of learning goal-directed behaviour can be reproduced by three signals exchanged between an agent and its environment, namely, the action signal to represent the choices made by the agent, the state signal to represent the basis on which the choices are made, and the reward signal to define the agent's goal.

2.5.2 Introduction to Reinforcement Learning

A popular approach to solve finite MDPs is Reinforcement Learning (RL). Essentially, RL involves an autonomous agent that interacts with an environment by performing certain actions on it, to reach a final goal. In practice, an RL algorithm learns what to do, by mapping situations to actions, in order to maximise a numerical reward function [57]. Interestingly, the learner is not provided with explicit instructions on which actions to choose, instead, it must explore and determine which actions result in the highest rewards through trial-and-error. In addition, actions have the potential to impact not only immediate rewards but also the subsequent state and all future rewards that follow. These two features, trial-and-error search and delayed reward, distinguish RL from other ML branches such as *supervised* and *unsupervised* learning [57].

In the following, we define some key elements of RL which were mentioned above:

- **Agent**: The RL agent is the entity that learns and makes decisions, it interacts with the environment, observes its state, and takes actions to influence it. For example, it can be an algorithm in a container deployed in a Kubernetes cluster, as well as the model that controls a robotic arm.
- **Environment**: The environment represents the external system with which the agent interacts. It can be a simulated environment in a computer program, such as a simulation of a Kubernetes cluster, or the physical world, as in the case of a robot that learns how to grasp real objects. The environment provides feedback to the agent through rewards or penalties based on its actions.
- **State**: The state of the environment represents the information that the agent uses to make decisions, called an *observation*. Its definition captures the relevant aspects of the environment at a given point in time. For example, to define the state of a Kubernetes cluster, one can include the number of replicas, the average CPU utilisation, or the input workload, among many other variables. In the case of a robotic arm, the state can include the position of its joints in a three-dimensional space.

- **Actions**: Actions are the choices available to the agent at any given state. The agent selects an action based on its policy and executes it in the environment. For example, two available actions could be to *scale-out* and to *scale-in* a K8s deployment, or to add/subtract a quantity from a vector of angles that control the rotation of the joints in a robotic arm.
- **Rewards**: Rewards are scalar values that the agent receives from the environment in response to its actions. These rewards serve as the feedback signal for learning, guiding the agent towards the final goal. It is crucial to design a reward function that incentivises the desired behaviour. For example, a very basic reward function would add 10 every time the agent gets closer to the goal and subtract 1 every time it moves away. The idea is to "reinforce" a good action in a specific state so that it will more likely be taken again in the future.

Lastly, RL problems can be *episodic* or *continuous* [57]. In episodic tasks, there is a *terminal state* that the agent can reach to end the episode, and can successively start another independent episode. For example, the end of an episode could be determined by a maximum number of actions taken during a time interval or the reach of a goal, such as the robotic arm that successfully grabs an object. On the contrary, in a continuous RL problem, there is no such notion as terminal state. The agent-environment interaction does not break naturally into episodes, rather, it is an on-going process-control task [57]. In this scenario, the concept of *discounted cumulative reward* is used, which will be introduced shortly.

2.5.2.1 Value Learning Methods

To expand on the idea of transition probabilities, we introduce the concepts of *policy*, to define the probability that the agent selects an action given a state, and of *value function*, to evaluate how good is the decision or the state.

A *policy* is a mapping from observed state $s \in \boldsymbol{S}$ to action $a \in \boldsymbol{A}(s)$ that defines how the agent should behave when in that state [57].

$$\pi(a|s) = Pr\{A_t = a|S_t = s\} \tag{2.3}$$

Therefore, for each state s, the policy π is a probability distribution over possible actions a in that state. Based on the reward, the agent updates its policy for that state.

Another important concept is the *value* of a state, which is the total discounted return G_t that the agent can expect to accumulate starting from that specific state. The total discounted return is defined as $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, where $\gamma \in [0, 1]$ is the *discount factor*, and it is used to favour immediate rewards over future rewards. Differently from the immediate reward, the *value function* specifies the quality of a state or an action in the long run.

There are two distinct types of value functions that are used in RL. The *state-value function* for policy π is the expected return of following the policy from state s, and it is defined as $v_\pi(s) = \mathbb{E}[G_t|S_t = s]$. On the other hand, an *action-value function*, tells how good the expected return is by following a given action in a given state for policy π, and it is defined as $q_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a]$, often referred to as *Q-function*.

The purpose of estimating values is to maximise the expected discounted cumulative reward over time G_t, known as *Q-Value*. For this reason, many RL methods, such as Q-Learning, State-Action-Reward-State-Action (SARSA), and Deep Q-Network (DQN), focus on learning the optimal value function, from which the optimal policy can be selected. That is, learning to select actions that lead to the most valuable states, which potentially bring the highest rewards in the following timesteps. In the case of $v_\pi(s)$, the optimal state-value function is $v_\star(s) = \max_\pi v_\pi(s)$ for all $s \in \boldsymbol{S}$, which is the largest expected return given by any policy in any state. In the case of $q_\pi(s, a)$, the optimal action-value function is $q_\star(s, a) = \max_\pi q_\pi(s, a)$ for all $s \in \boldsymbol{S}$ and $a \in \boldsymbol{A}(s)$, being the largest expected return achievable for any state-action pair for any policy.

Lastly, value based methods such as Q-learning are advantageous because of their sample efficiency. This means that they require relatively few examples to learn an optimal policy because they can learn from data generated by a different policy than the one being updated, even from synthetic data generated by experts [34].

2.5.2.2 Policy Gradient Learning Methods

In action-value methods the agent learns the value of an action and then selects the actions based on their estimated action values [57]. This allowed to have a mapping $\pi : s \rightarrow a$, which returns an action a based on the state s, where the policy is updated based on the estimated return computed with the value function. Instead of learning the value function for the estimated return, *policy gradient methods* aim at learning directly a parametrised policy based on the gradient of some performance measure, without estimating the values [57]. This approach offers some advantages compared to value-based methods, which will be mentioned shortly.

In this scenario, the way in which the policy is updated without information about the value of a state or state-action pair, is to define it in terms of an external parameter $\boldsymbol{\theta}$ as

$$\pi(s, a|\boldsymbol{\theta}) = Pr\{A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\} \tag{2.4}$$

where $\boldsymbol{\theta}$ is a vector of parameters. Thus, the policy for state s at time t, given a parameter $\boldsymbol{\theta}$ at time t, is the probability distribution over the possible actions given by s and $\boldsymbol{\theta}$. Note the difference with respect to Equation 2.3, where the probability distribution is defined solely in terms of states and actions.

Therefore, in policy based methods, the target of the agent is to learn a policy parameter based on the gradient of a scalar performance measure $J(\boldsymbol{\theta})$, and update it with Stochastic Gradient Ascent (SGA) as $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)}$, where $\widehat{\nabla}$ denotes an approximation of the gradient. For example, the performance measure can be defined as the average rate of reward per timestep [57], $J(\boldsymbol{\theta}) = r(\pi)$. Specifically, it is possible to optimise for the policy parameter because the *policy gradient theorem* provides a way to compute the gradient of the performance measure with respect to $\boldsymbol{\theta}$ as

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \\ &= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta}) \right]\end{aligned} \tag{2.5}$$

as long as the policy is differentiable with respect to the chosen parameter, which in fact can be any parameter [57].

For example, such parameter can be the vector of weights of an ANN network used for function approximation. Briefly, ANNs are software computational models whose architecture and functioning are inspired by biological neural networks. They consist of layers of interconnected neurons that receive input data, each neuron multiplies its input by its own coefficient called *weight*, and returns an output filtered by an *activation function*. These networks are trained by processing many input examples, and extrapolate general features from the training dataset by adjusting the network weights as more data is fed to the model, with the goal of approximating non-linear functions and making predictions. In fact, these neurons are able to generalise the input data to similar datasets and are the fundamental mathematical algorithms applied in Deep Learning, a branch of ML. In the context of RL, when policy functions are stochastic, meaning that the sum of all numbers adds up to 1, it is possible to approximate the functions with an ANN to reach a deterministic policy [57]. This approximation approach exploiting Deep Learning algorithms such as ANNs is called Deep Reinforcement Learning (DRL), and we will make use of such paradigm in this thesis.

Policy based methods provide several benefits. For example, one advantage of parameterizing policies is that the approximate policy can approach a deterministic policy, whereas with ϵ-greedy policies with action values, there is always an ϵ-probability of selecting a random action [57]. A second advantage of parameterizing policies is that, in problems requiring significant function approximation where the best approximate policy may be stochastic, it enables the selection of actions with arbitrary probabilities, rather than being constrained to deterministic actions or fixed probabilities. [57]. Moreover, sometimes it is easier to approximate the parameters of a policy rather than those

of a value function [57]. In essence, high-dimensional action space handling capabilities and function approximation are key features that make policy gradient a popular RL technique.

2.5.3 Actor-Critic in Reinforcement Learning

When it comes to balance trade-offs between different metrics in autonomous container orchestration, model-free RL algorithms such as Actor-Critic, are potential candidate solutions [71]. In fact, since the release in 2016 of Asynchronous Advantage Actor Critic (A3C) [33], Actor-Critic methods gained popularity and have recently been employed in container orchestration problems with relative success [6], [48], [68].

In essence, the Actor-Critic algorithm combines a hybrid architecture of policy-based and value-based methods, for the *actor* and *critic* respectively, with the aim to exploit the advantages of both methods to help stabilise the training phase by reducing the variance [6]. In fact, the Actor-Critic algorithms leverage the stability and convergence guarantees of value-based methods, while benefiting from the exploration features and approximation capabilities of high-dimensional action spaces with policy-based methods. Specifically, the A3C algorithm is a DRL framework that uses asynchronous gradient descent for optimisation, which is conceptually simple and lightweight compared to other DRL methods [33].

As illustrated in Figure 2.5.2, the agent incorporates an Actor ANN, which interacts with the environment, and a Critic ANN. Both Actor and Critic observe the environment in its states, which become the input features $x_1, x_2, ..., x_n$ for both ANNs. Based on the observation, the policy-based network takes an action based on the policy probability distribution $\pi(A_t|S_t, \boldsymbol{\theta}_t)$ across all actions $A_1, A_2, ..., A_n$. The parameter $\boldsymbol{\theta}$ is the parameter of the policy learned by the agent and, in the case of an ANN, $\boldsymbol{\theta}$ is a vector of network weights. The value-based network observes the reward from the environment, and computes the Temporal Difference (TD) error δ with respect to the value V of the previous state. TD refers to the technique of updating value estimates by bootstrapping from other value estimates, using the difference between successive estimates at different timesteps [57]. The TD error is then used to update the ANN weights of both actor

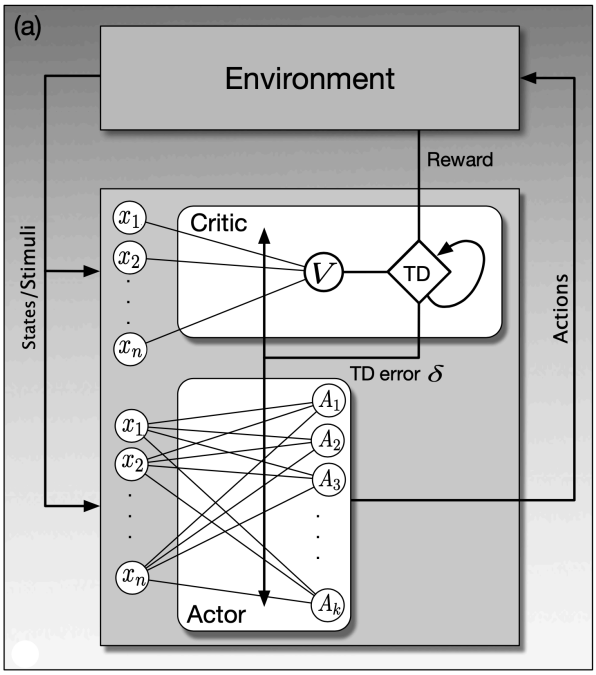


Figure 2.5.2: Basic implementation of an Actor-Critic with ANNs for function approximation.
Source: [57]

and critic networks. Therefore, the output of the value network is an estimate of $v_\pi(S_t)$, while the output of the policy network is a policy distribution over placement actions [6].

2.5.3.1 Advantage Actor-Critic (A2C)

In our implementation, we used a synchronous version of A3C called Advantage Actor Critic (A2C) [37], implemented by OpenAI in a Python library called `Stable Baselines3` [9], [54].

The *advantage* is defined as $A(S_t, A_t) = q_\pi(S_t, A_t) - v_\pi(S_t)$, and it is the quality measure that is used to update the ANN weights [33]. The basic idea behind A2C is to reinforce an action if its advantage with respect to the average quality of all possible actions in that state is higher. With the variant of the REINFORCE algorithm with baseline [33], a popular policy gradient algorithm, the gradient of the parametrised

policy function becomes

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}\left[\sum_{a}(q_{\pi}(S_t, a) - v_{\pi}(S_t))\nabla \ln \pi(a|S_t, \boldsymbol{\theta})\right] \tag{2.6}$$

where the baseline is the value function computed by the critic. The purpose is to ensure a much lower variance in the estimation of the policy gradient, such that policy learning is more stable [6].

The feedback loop between Actor and Critic involves the following steps.

1. **The Actor** (policy network): The actor component takes in the observations as inputs, for example *CPU utilisation* and *latency* as states of a K8s deployment, and generates actions based on the current policy. In this case, the output is obtained by passing the processed features through the policy network, which consists of two fully connected layers followed by activation functions.
2. **The Critic** (value network): The critic component estimates the value function, which represents the expected cumulative reward from the current state. It takes in the same observations as the Actor and outputs an estimate of the state value using the value network, which also consists of two fully connected layers with respective activation functions.
3. **Value estimation**: The output of the value network represents the estimated value of the current state. It provides a baseline for evaluating the advantage of the chosen actions.
4. **Advantage estimation**: The advantage is calculated by taking the difference between the estimated value and the actual observed rewards. It represents how much better or worse the chosen action is, compared to the expected value.
5. **Actor update**: The actor is updated based on the advantage estimation. The update aims to improve the policy by adjusting the action probabilities to maximise the advantage. This weight update occurs using Stochastic Gradient Ascent.
6. **Critic update**: The critic is updated based on the difference between the estimated value and the observed rewards. This update step aims to minimise the discrepancy between the

predicted values and the actual rewards using TD methods.

During the training phase, the Actor-Critic model is deployed in the environment and by interacting directly with it, it learns the optimal policy. In the learning phase, both networks update the weights of their neural connections in a way that maximises the total discounted reward. Once the training phase is completed, the model is saved and can be used to make inferences in similar environments.

2.6 Related Work

In this section we present the state-of-the-art in container orchestration, specifically to autoscaling with ML, and studies on the HPA with a multi-metric approach. We begin by providing the general context in container autoscaling with a systematic literature review [71] and a survey on autoscaling in Kubernetes [61]. We then present works specifically targeting multi-metric approaches [11], [36], [69], [72]. Then, we present studies that employed RL for automatic orchestration of containers [49], [50], [59], [70], and finally we write about the research on Actor-Critic models used in container orchestration [6], [48], [68], highlighting the need to further research the state space definition's influence on the performance of those models.

2.6.1 State-of-the-art in container orchestration

In recent years, cloud service providers and most of their clients have been using container technologies in their distributed system infrastructures for the automatic management of their applications. However, the automation of deployment, maintenance, autoscaling, and networking of containerised applications, make container orchestration a serious research problem. Under such complex and dynamic environments, a promising approach is to employ ML algorithms to improve the quality of resource provisioning decisions by modelling and predicting the system across multiple dimensions. A comprehensive literature review on such approaches was published in 2022 [71]. Here, the authors propose a detailed taxonomy to classify the state-of-the-art, and highlight future research opportunities in the field.

With respect to the taxonomy, the authors grouped the literature based on their focus on *application architecture* (monolithic, microservices, serverless), *infrastructure* (single cloud, multi-cloud, hybrid cloud), *optimisation objectives* (resource efficiency, energy efficiency, cost efficiency, SLA assurance), techniques such as *behaviour modelling and prediction* (workload characterisation, performance analysis, anomaly detection, dependency analysis) and *resource provisioning* (scheduling, scaling, migration). Specifically, in the scope of this thesis, the areas of interest are *MSA* in *single cloud* environments, with *SLA assurance* and *cost efficiency* as optimisation objectives, and resource provisioning with *scaling* technique as strategy to solve such challenges. As mentioned above, scaling refers to the size adjustment of containers or computational nodes in response to demand fluctuation, with the aim of ensuring that the containerised applications are given enough resources to minimise SLA violations. In their taxonomy, the authors further classify scaling as *horizontal*, which is adjusting the number of containers and is in fact the subject of this thesis, *vertical*, which is scaling by updating the amount of resources like CPU, memory or storage given to an existing container, and finally a *hybrid* version that combines both solutions.

Regarding ML-based approaches, the authors outline the evolution of such models in the period from 2016 to 2021. And, for what concerns resource provisioning, the most recent solutions employed RL for decision-making, and a combination of multiple existing ML methods to create a complete container orchestration pipeline. The following list resembles the timeline proposed in [71] for resource provisioning approaches:

- In 2019, there was the first application of DRL for task scheduling [6]. The goal was to design an algorithm for the co-placement of batch processing jobs, and the researchers employed an Actor-Critic algorithm to train an ANN model to generate optimal scheduling decisions. Compared to traditional heuristics, their solution provided remarkable improvements in terms of overall job execution time.
- In 2020, [49], [50] leveraged model-based RL for hybrid autoscaling in monolithic applications, aiming at a cost-effective solution that could minimise performance degradation. In the same year, more hybrid algorithms were proposed, which use

a combination of RL-based solutions for decision-making and ML models for timeseries prediction and dependency analysis. For example, in [48] the authors propose *FIRM*, an Actor-Critic method used to prevent Service Level Objective (SLO) violations and generate optimal scaling decisions. Another example is *A-SARSA* [70], which used a workload predictor to provide an RL agent using Q-Learning with the states on which to compute the scaling policy. These combinations proved to perform better than stand-alone RL solutions.

- In 2021, *HANSEL* [67] was proposed to improve the speed and efficiency of RL-based scaling approaches in MSA by combining Bi-directional Long Short Term Memory (Bi-LSTM) for load prediction with parallel training of RL agents.

As highlighted in this analysis, the emerging trend is to combine multiple ML methods and focus on end-to-end solutions that can form a complete orchestration pipeline. In addition, looking at the most recent years, [71] points out that model-free RL algorithms are the current benchmark for scaling with multiple metrics and, in particular, Actor-Critic algorithms are potential solutions when it comes to balance trade-offs between different metrics. Finally, as a general remark on machine learning-based methods applied in container orchestration, the authors observed that there is still no systematic approach to build a complete ML optimisation framework, nonetheless, these frameworks are crucial as the system complexity of container orchestration tasks grows.

Nowadays, the de-facto standard tool for container orchestration is Kubernetes, which is the reason why many researchers in the field have focused on this software. Regarding autoscaling in Kubernetes, [61] aims at presenting the state of the art of the current approaches. In particular, the authors categorise existing solutions based on *application architectures* (monolithic, MSA), *methods* (horizontal, vertical, and hybrid scaling), *timing* (reactive, proactive scaling), and *performance indicators* (low-level metrics, high-level metrics). Based on these distinctions, the authors discuss the issues with the current autoscaling methods in Kubernetes.

In particular, regarding performance indicators, the authors distinguish between *low-level* metrics and *high-level* metrics. The first

category comprises physical resource indicators such as CPU, memory, network input/output, and disk, although CPU and memory are the most adopted in the industry. These metrics can be used to directly trigger the autoscaling action. On the contrary, high-level metrics are application-level performance indicators, for example, traffic or workload rate, request/response latency, and SLO. However, these are used as support to take efficient scaling actions, as they cannot be used to directly trigger the autoscaler.

Since the default HPA depends on manually setting the scaling thresholds values, in particular the CPU utilisation [5], there have been attempts from researchers to provide a better mechanism. Specifically, [61] reports studies focused on alternative scaling metrics [49], some of which employ heuristics and ML to customise the HPA [59], [69]. These studies will be analysed in the following sections.

In summary, the focus of recent research is on leveraging machine learning to improve container orchestration, with an emphasis on end-to-end solutions and resource provisioning through scaling techniques. The adoption of ML algorithms, reinforcement learning in particular, is becoming prevalent in making resource provisioning decisions. In the context of Kubernetes, efforts are being made to enhance autoscaling mechanisms by considering different metrics and employing ML and heuristics-based approaches.

2.6.2 Alternative approaches to CPU utilisation as scaling metric

In this section we present researches that have studied the influence of alternative metrics, other than CPU utilisation, in Kubernetes clusters. In particular, we report a first study that discusses the scaling differences when using the HPA with the default Kubernetes Resource Metrics, able to provide only metrics concerning CPU and memory, and Prometheus Custom Metric, used to fetch much richer metrics from the cluster [36]. A second study proposes *service power* as an alternative latency-based metric used to scale because the authors argue that the correlation between performance degradation and CPU utilisation is sometimes weak [69]. Another study explores the use of relative and absolute CPU values when scaling in Kubernetes [11]. Finally, a fourth study combines the use of CPU utilisation with a queue length metric

to trigger the autoscaling action and shows performance improvements [72].

In [36], the authors perform seven different experiments with the aim of a detailed understanding of the HPA's behaviour. In particular, in the scope of this thesis, their experiments on the HPA with a combination custom metric are our focus. The authors test the autoscaler when it is triggered by metrics sourced from the `kube-metrics-server`, which in the paper is referred to as *Kubernetes Resource Metrics* and provides only CPU and memory measurements, and when alternative metrics are scraped from Prometheus. Firstly, the authors test the HPA with the HTTP request rate only as custom metric, and in a second experiment the authors combine it with the average CPU utilisation. When multiple metrics are specified, the scaling action occurs if either one of the metrics reaches its specified threshold. In their experimental setup, the authors employ a CPU-extensive application deployed in Kubernetes, which receives a step-wise load of synthetically generated HTTP requests from a loadgenerator. To measure the success of the scaling action the authors consider the number of failed requests by the application. From their experiments, the authors conclude that a combination of custom metrics, in their experiments average CPU utilisation and rate of incoming HTTP requests, can be beneficial for the HPA's effectiveness because changes in any of the individual metrics would cause the scaling action. However, the triggering value have to be carefully set, otherwise it may lead to a waste of resources. Therefore, the authors remark the fact that combinations of metrics should be chosen according to the type of application that is considered. For this reason, we decide to consider multiple metrics in out experiments.

Another interesting approach to improve the autoscaler by considering other metrics is *Microscaler* [69]. In their research, the authors propose a cost-effective solution combining Bayesian Optimisation with a step-by-step heuristic model to detect the scaling needed microservice and scale it to meet the SLA requirements. In particular, the authors select the *service request latency* to measure the application performance instead of the resource utilization or the request volume because it is directly related to the SLA constraint on response time. This is motivated by the fact that the correlation between CPU utilisation and QoS may not be strong, and that in MSA

it is crucial to identify which microservice is causing the SLA violation by performing dependency analysis. In fact, the authors show that if the average CPU utilisation of a pod is high, the overall response time of the application may still be acceptable. For these reasons, the authors introduce *service power* as performance metric, defined as the ratio between the p_{90} and p_{50} latencies, and use this to detect the scaling-needed microservice. Being p_{90} the latency of the slowest 10% of responses averaged in the past $30s$, and p_{50} the average latency of the slowest 50% of the responses generated by the microservice in the past $30s$. To compute the cost of a scaling action, the algorithm considers the number of replicas and the average latency of the deployment. In their experiments, the authors use an early version of the benchmark application *Online Boutique* deployed in a Kubernetes cluster, and adopt the service mesh framework with Istio to track latency metrics via Envoy proxies. The authors set as SLO a response time T_{max} of $2s$ and a maximum number of replicas for each instance to 15. Specifically, the authors compare *Microscaler* with a horizontal autoscaler set at 70% average CPU utilisation when a step-wise workload pattern is produced. In particular, the authors show that traditional methods only using CPU or memory to understand the scaling needed-service are not precise compared to *Microscaler*, which leads to unnecessary scaling. Lastly, the proposed solution is able to find in a few iterations the optimal number of replicas for the microservice responsible for the SLA violation. In this thesis, we will use the same benchmark microservice application to test our results, with the number of replicas for each deployment in the interval $[1, 15]$.

In [72] the authors propose a bi-metric autoscaling strategy for HTTP servers in Kubernetes, by combining CPU utilisation with the utilisation of container's queue length. Specifically, the authors address two limitations of the standard HPA, namely, the reliance on solely CPU utilisation, and the load imbalance caused by stateful HTTP requests. Although the second limitation has been addressed by the Kubernetes community with the introduction of *StatefulSets* in K8s version `1.15` [51], while the authors were using version `1.13.4`, the former is of interest for our research. In fact, the authors compare their approach with the default HPA set at 80% average CPU utilisation and demonstrate that they could reduce the average utilisation of CPU by 14% while improving the response time of requests by 80%. Even

though the improvement of the response time can be largely attributed to solving the load imbalance problem caused by stateful HTTP, their experiments show that the average resource utilisation is affected by the two thresholds introduced with the bi-metric approach. To conclude, their study shows that a correct tailored parametrisation of autoscaling metrics, combining a hardware metric with an application metric such as thread pool length, can save resources on HTTP applications deployed in Kuberentes.

Lastly, an early study on the impact of CPU utilisation as scaling trigger for the Kubernetes HPA is [11]. Here, the authors explore the impact of relative and absolute values of CPU, the first being the default approach in Kubernetes, while the second is used in *KHPA-A*, the algorithm proposed by the authors. In this context, *relative* refers to metrics collected at the pod level by *cAdvisor*, which represents the share of CPU used by a container with respect to other containers, and it is expressed as the percentage of the total CPU given to the Control Group. On the other hand, *absolute* values are collected at the VM or infrastructure level, and represent the cumulative activity of the process counted in the OS. In fact, when tested on a simulation comparing the standard HPA with *KHPA-A*, the authors show that for high-loaded servers, the use of an absolute CPU value to trigger the autoscaling action results in a slower response time of the replica. Even though their simulation appears to be elementary in the computation of the response time and correlation between the relative and absolute CPU utilisations, it is interesting to see a tentative of overcoming the limitations of the default HPA.

2.6.3 Reinforcement Learning in container autoscaling

With respect to autoscaling with RL, the literature presents a few different approaches and in the following we report a selection of them based on their affinity with our work.

In [49], the authors compare model-free and model-based RL policies with the default threshold-based K8s HPA to scale applications with monolithic architecture. The ultimate goal of the solution is to meet QoS requirements in terms of the average response time of the application. In their results, the authors show that their model-based

RL solution, which is created based on what is known about the system dynamics, performs better than the other two approaches. Specifically, their RL models use the state defined as $\boldsymbol{S} = (k, u)$, where k is the number of pods and u is their monitored CPU utilisation. The possible actions taken by the agent are $\boldsymbol{A}(s) \subseteq \{-1, 0, +1\}$, where $+1$ defines a scaling-out action, -1 a scaling-in actions and 0 the *do nothing* decision. The reward is provided based on the sum of a performance penalty c_{perf}, paid whenever the average application response time exceeds the target value, a resource cost c_{res}, which is proportional to the number of running pods, and finally an adaptation cost c_{adp}, inversely proportional to the number of pods. The QoS is expressed as a maximum latency threshold $R_{max} = 80ms$. In their results, the model-free agent takes too much time to learn a good adaptation policy, and during the experiment time of 350 minutes, it violates R_{max} 64.0% of the times, compared to 14.4% of model-based and 9.20% of the HPA set at 60% average CPU utilisation. In particular, in our study we will use the same set of actions, the reward based on latency, and the HPA set on average CPU utilisation. As a final note, this study is only concerned with satisfying the QoS and does not care about overprovisioning of resources, while in our work we are trying to optimise also for this aspect.

The same authors of [49], in a later work propose *me-kube* (Multi-level Elastic Kubernetes), a Kubernetes component combining a heuristic based on queuing theory and Autoregressive Integrated Moving Average (ARIMA), a statistical method for timeseries forecasting, to perform proactive horizontal autoscaling [50]. In their novel paper, the authors focus on MSA while still targeting latency-sensitive applications and keeping as requirement the maximum response time R_{max}, which should not be exceeded. The approach uses a hierarchical autoscaling, which consists of a centralised *Application Manager* control loop in charge of coordinating the adaptation of the scaling policy for the global application, and decentralised *Microservice Managers* control loops, in charge of the individual microservices. The hierarchical solution is then compared with the fully decentralised model-based RL solution in [49] and with the default Kubernetes HPA. From their results it emerges that the proposed heuristic performs better than the two other solutions. In fact, the major disadvantage of the fully decentralised model-based RL solution was that the lack

of coordination between agents lead to unnecessary reconfigurations. On the other hand, with a hierarchical policy, there was no such problem as the *Application Manager* could choose which microservice to scale based on the general knowledge about the application. In addition, the prediction of the RPS coming to the services proved to be beneficial when choosing the optimal policy for the heuristic method. Finally, even though a fully decentralised solution may have lower performance that the heuristic, in fact the former exceeds R_{max} 46.20% of the time while the latter only 5.56%, in our work we decided to adopt a decentralised solution by deploying one agent per deployment because we are interested in studying the effect of the state space definition on the agent's action, rather than proposing an end-to-end scaling solution.

An improvement on [49] is presented in [70], which introduces *A-SARSA*, an algorithm that integrates a *workload prediction* component with a *state prediction* component, to overcome the limitations of (1) having a state-action-reward model such as Q-Learning and (2) a fixed action space of 3 actions $\boldsymbol{A} \subseteq \{-1, 0, +1\}$. In fact, the authors point out that the state-action-reward model does not take into account the fact that the state not only is affected by the agent's action, but also by the changing workload. For this reason, instead of Q-Learning the authors adopt *A-SARSA*, a modified version of SARSA, which traditionally updates the $Q(s, a)$ value based on the action chosen in the next state S_{t+1}, while in *A-SARSA*, S_{t+1} is the result of a prediction model. Their *workload predictor* uses the ARIMA model to take as input the current RPS, and output the predicted RPS. This result is then used by the *state predictor* component, which is an ANN that takes CPU utilisation, memory utilisation, response time, RPS, number of replicas, and the predicted RPS as inputs, and predicts the CPU utilisation α and response time β of the service. This output is used as state space $\boldsymbol{S} = (\alpha, \beta)$ for the modified SARSA agent to take the next action A_{t+1}. In addition, to address the limitation of a fixed state space, which may occur when there is a sharp increase in the workload and only adding one replica is insufficient, the authors iteratively increase the action space $\boldsymbol{A}$. For this reason, in *A-SARSA* the algorithm symmetrically adds one action if it is deemed necessary in that state, for example with $\boldsymbol{A}(s) \subseteq \{-2, -1, 0, +1, +2\}$. Regarding the reward function, the authors design one that avoids SLA violations and maximises the

CPU utilisation, which, as opposed to [49], introduces the resource optimisation objective. Finally, with their results, they show the superiority of *A-SARSA* compared with Q-Threshold, SARSA and model-based RL in terms of SLA violation rate, which they attribute to the predictive feature of the algorithm. Although their approach aims at creating a performative end-to-end autoscaling pipeline, we will nonetheless adopt some of their implementation designs in our work. Specifically: the use of the 95th percentile response latency p_{95} as SLO, the idle period of $40s$ between actions to ensure that the collected metrics are not affected by actions of the previous cycle, a modified version of their reward function, a benchmark microservice application for the experiments, and the monitoring architecture using Istio service mesh, Envoy proxies, Prometheus and Grafana in a Kubernetes cluster.

One of those solutions that combined multiple ML methods to achieve scaling accuracy is *HPA+* [59]. In their research, the authors introduce a customisation of Kubernetes HPA that combines the predictions of four ML models on the forecasted load, and computes the number of required pods accordingly. Moreover, the number of pods is further corrected with a new proposed parameter, called *excess*, which is used by the cloud tenant to set the trade-off between resource over- and underprovisioning to balance SLA violations and costs. The algorithms the authors employed were an Auto-Regressive (AR) model, two artificial neural network models, namely Long Short-Term Memory (LSTM) and Hierarchical Temporal Memory (HTM), and an RL-based approach. The authors show that, individually, these models do not always provide satisfactory results, which is the reason why they decided to combine the four forecasting tools to produce the optimal output. Essentially, their algorithm takes as input four forecasted values of traffic, one from each model, and the most accurate one is selected to compute the required replicas. The authors show that *HPA+* achieves up to 72% less lost requests, at a price of additional 9% resource usage, compared with the default HPA scaling on CPU, though they do not report the threshold value used in the latter. With respect to their RL model, the authors used SARSA and Q-Learning, where the states given to the models were (1) the current load and (2) the number of running pods. However, in their experiment, the authors show that the prediction accuracy of the RL-based approach

is always lower than other models, and therefore its output was never used. This corroborates the hypothesis that a combination of ML solutions is beneficial for a complete container orchestration pipeline [71], especially considering more advanced meta-learning forecasting methods [21], and that standard SARSA or Q-Learning alone are limited solutions [70]. Lastly, similarly to the *excess* measure introduced in [59], in our work, we give cloud tenants the possibility to regulate the optimisation degree.

2.6.4 Actor-Critic methods in container orchestration

As highlighted in [71], Actor-Critic methods have recently been tested to balance the trade-offs between different metrics in container orchestration. In the following, we report three works that employed such RL architecture and have evaluated it in Kubernetes. The first, *Harmony* [6], has been used to optimise job co-location in ML clusters with model-free RL. The second, *MIRAS* [68], optimises resource allocation in microservices with model-based RL. The third, *FIRM* [48], employed an Actor-Critic network to provide optimal resource assignments to mitigate SLO violations.

In [6], the authors designed *Harmony*, a DRL framework for the placement of ML training jobs in clusters, with the goal of reducing the training time by minimising interference. Their results show a 25% decrease in average job completion time compared to other representative scheduling policies. Basically, in cases of parallel training for large ML models, the problem of ML job interference in clusters arises when multiple jobs running on the same machine compete for shared resources, leading to performance degradation and slower training times. The goal of *Harmony* is to autonomously learn how to assign newly arriving jobs to the best machine based on a state space comprising

1. a binary matrix encoding the ML models trained by the jobs;
2. a matrix encoding the machine resource demands in terms of CPU and Graphical Processing Unit (GPU) cores;
3. an N-dimensional vector, in which the n^{th} item is the number of workers allocated to the n^{th} job;

4. a matrix representing the available amount of each type of resources on the machines;
5. a matrix encoding the concurrent placement of jobs in the machines;

The action taken by the agent is executed during a scheduling interval when multiple new ML jobs may arrive. Therefore, the agent has to find the optimal placement of a worker in a machine for that job, among all possible combinations of all workers and all jobs. Since such problem can be formulated as sequential decision-making in an unknown environment, but where both state and action spaces are very large, to solve the task the authors adopt DRL, a technique combining RL with an ANN to approximate value functions or policies. As reward, the authors use the sum of normalised training speeds of all concurrent jobs in the scheduling interval, which has to be minimised. In addition, to stabilise the training of the DRL algorithm, and ensure quick convergence to an optimal policy, the authors employed the Actor-Critic algorithm to reduce the variance in the Q values. Finally, compared with traditional heuristic scheduling policies like bin packing, their solution demonstrated remarkable performance improvement on the Kubernetes cluster regarding overall job execution time.

To address the limitation of high sample complexity of model-free RL approaches, the authors of [68] proposed *MIRAS* (Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows). To overcome such limitation, which would make the solution hard to motivate in real-world scenarios, the authors adopt a predictive model of the environment to further train the agent. Specifically, in their architecture, the authors model each task type as a microservice consisting of a request queue and a set of consumers subscribing to the queue to handle requests, and their optimisation objective is to minimise the average processing time of each task taken by a microservice. To do so, the authors employ a model-based Actor-Critic RL solution for policy learning to make decisions on microservice resource allocation. In their RL problem, the state space is defined as the delay of a task in a microservice $w(k)$ in the k^{th} time window, which is measured as the time elapsed from when the task arrives in the microservice message queue to the time it leaves the microservice after being processed. The action $m(k)$ taken by the agent is the

number of consumers that can be allocated for a service, therefore, a horizontal scaling action. Finally, the reward function $r(k)$ is defined as the negative of the sum of all ongoing tasks across microservices observed at the end of each time window, which have to be minimised. Regarding their algorithm implementation, the authors adopted the Deep Deterministic Policy Gradient (DDPG) method available within the OpenAI's Baselines Python library. In particular, for this degree project, we will make use of an updated version on the same library because it provides ready-to-use models and allows us to focus on our research questions rather than on the implementation of the DRL algorithm itself. To conclude, the authors validate the performance of *MIRAS* against other existing workflow management algorithms, including a model-free DDPG. In particular, they show the superiority of *MIRAS* compared with its model-free version lacking the predictive feature, which does not converge to a good policy in time due to its poor sample efficiency.

Another end-to-end ML container orchestration pipeline for microservices developed with DDPG is *FIRM* [48]. Specifically, *FIRM* is a framework used to manage shared resources among microservices with fine granularity to increase resource utilization and minimise SLO violations in terms of end-to-end response latency. To achieve their goal, the authors employ a dependency analysis module that detects the bottleneck microservice causing SLO violation and an Actor-Critic RL module to generate the appropriate resource assignment decisions through vertical scaling. The approach has been validated and executed on the Kubernetes cluster with significant performance improvement compared with Kubernetes autoscaling. In their problem formulation, each microservice is deployed in a separate container with pre-defined resource limits, usually overprovisioned. At each timestep t, the RL agent observes a state consisting of

1. resource utilisation RU_t at the container level, such as CPU and memory utilisation, and at the node level, like Last-Level Cache (LLC) capacity, disk input/output bandwidth and network bandwidth;
2. SLO maintenance ratio SM_t, defined as the ratio between the SLO latency and the current measured latency;
3. workload changes WC_t, as the ratio between the rate of requests

arrival between the current and the previous timesteps;

4. request composition RC_t, as an encoded array representing the precentages of each type of request.

Based on these states, the action taken consists in setting the new request limits for the resources. The reward function is designed in such a way that the agent learns to have the least amount of SLO violations while keeping resource utilisation as high as possible. The authors train *FIRM* on a Kubernetes cluster with a benchmark microservice application and validate on another microservice application to test the generality of the learning. Their results show that the combination of the ML model to identify the microservice responsible for SLO violation with the RL model for mitigation, reduces SLO violations up to 16 times while reducing the overall CPU limit by up to 62% compared to the Kubernetes autoscaler.

2.7 Summary

Cloud computing is an established industry paradigm that allows clients to rent computing power instead of building and maintaining their own expensive data centres. It is characterised by features such as scalability, elasticity, and security. The main technologies that drive its functioning are the Internet and virtualisation.

Nowadays, software containers such as Docker are used in cloud computing platforms to accommodate for the scalability and isolation that software applications need. These applications are often deployed in the microservice architecture, which is a framework with the aim of making the development and deployment of such applications faster, by segmenting them into smaller, independent units. However, the speed of development comes with a cost of increased complexity, and container orchestration tools such as Kubernetes are employed to manage microservices packaged into containers, along with service mesh and monitoring tools such as Istio and Prometheus.

In particular, to handle the scalability of containers, called pods in K8s, Kubernetes makes use of the HPA based on a trigger performance metric. By default, this metric is the CPU utilisation of the pod, and the

triggering value is manually tuned. The problem with this approach is that relying solely on this metric to measure the performance of all microservices often leads to over- or underprovisioning of resources.

There have been research works investigating alternative metrics, some employing RL in container orchestration, and specifically Actor-Critic algorithms for end-to-end container orchestration. Such machine learning algorithms are able to learn what do to without prior knowledge with an iterative process of interaction with the cloud environment, observation, and learning from rewards. However, what is missing is a study on the influence of such alternative metrics in the decisions of the algorithm, from which originates the main research question of this thesis.

Chapter 3

Method

This chapter presents the engineering methodology that motivates the design choices and the planned measurements intended to answer our research questions. Our aim is to assess the influence of different state space definitions in a model-free DRL agent's ability to find the optimal autoscaling policy in a Kubernetes cluster. In the following, we provide a motivation for our major design choices, a mathematical problem formulation, the experimental setup and its designed architecture, and the motivations for the designed experiments.

3.1 Research process

The aim of this research is to contribute to the field of autonomous container orchestration, specifically, by providing empirical results on the state space influence on a model-free DRL agent's performance in taking optimal scaling actions in a Kubernetes cluster. In particular, given the fundamental relationship between policy, actions, and states in an RL problem formulation, we expect to measure a positive correlation between the state space and the optimal policy identification for horizontal scaling.

The steps conducted to carry out this research can be broken down into the following list:

1. **Literature review**. The preliminary step was to assess the state-of-the-art in autonomous container orchestration, in particular, with ML. We started querying publication portals

for keywords such as *machine learning*, *Kubernetes*, *container orchestration*, *optimisation*, *metrics*, and refined our selection of keywords as our understanding of the research panorama improved, eventually querying for the final set of keywords reported in the Abstract of this document. Then, starting from two comprehensive literature reviews on the topic, we analysed the most relevant papers referenced by these, until we believed the most relevant research have been read. Lastly, we tried to select our bibliography based on its publication from authoritative sources.

2. **Research question**. Confident that [71] has systematically assessed the state-of-the-art in container orchestration with ML from 2016 to 2021, supplemented with our literature review by keyword search targeting RL approaches with alternative performance metrics, plus the selection of additional resources from the bibliography of referenced papers reported in Section 2.6, we identify the research gap as described in Section 1.1. Thus, the formulation of our main research question as: *What is the impact of including specific container's performance metrics (e.g. CPU, memory, latency, RPS) in the state space on a model-free RL agent's ability to accurately predict optimal autoscaling actions?*

3. **Choice of the DRL framework and architecture design**. To answer our research question empirically, we had to (i) deploy in a Kubernetes cluster an agent capable of taking scaling actions, and (ii) study its behaviour with different state space definitions. Therefore, in our architectural design, we adapt an existent DRL model to our custom case, and deploy the microservice application *Online Boutique* in a GKE cluster, on which we perform our measurements.

4. **Design of the experiments**. We proceeded by designing a series of experiments that would help us answer our research questions. We carried out both simulations and emulations to model our problem and fit our needs.

5. **Measurements and conclusions**. Lastly, we gathered the data we needed from the emulation and used it to design a simulation of the environment that could be used by the agent

to improve the speed of its training. We completed our research by further analysing the results and draw our conclusions from the data.

3.1.1 Motivations of major design choices

- **Cloud native software**. We decided to emulate our cloud system to gather empirical data. For this reason, we employed specific cloud software available in the market, most of which are free and open source. Specifically, we used Kubernetes as container orchestrator because it is currently the industry standard [50], and, in particular, we exploited the Google managed Kubernetes cluster GKE. Rather than creating a microservice application from scratch, we took a benchmark application with MSA called *Online Boutique*, as it has been used in other research for testing [69]. Our monitoring stack included popular open-source software for cloud environments such as Istio Service Mesh, Prometheus, and Grafana, because they integrate well in Kubernetes, and have relatively good documentation.
- **DRL framework and A2C**. As our challenge inherently involves a goal-oriented online learning problem, we restricted our methods to reinforcement learning, instead of other ML methods. Specifically, because the state-action space may be very large when we consider multiple performance metrics in a highly dynamic cloud environment, we opt for a DRL framework for its ability to deal with approximations of stochastic policies [57]. The choice of Actor-Critic methods was picked as a consequence of our understanding of the state-of-the-art in container orchestration with ML [71]. Lastly, we implemented our Actor-Critic DRL agent with model-free A2C for its relatively low computational requirements compared to other models of OpenAI's `Stable Baselines3` library, its ability to handle discrete actions spaces, and good performance on CPU [33], [54], as GPUs were not available for our implementation.
- **Reward function**. The desired behaviour of the agent is to learn an optimal policy approximation that allows to scale a deployment in such a way that (i) the SLA of maximum response

time is not violated, and (ii) the number of replicas is minimised. The rationale behind this double constraint is to avoid that the agent learns to comply with the SLA by always overprovisioning with the maximum number of available pods, resulting in a waste of resources and increased monetary cost. Moreover, the SLA on latency has been used in several researches we analysed.

- **Performance metrics**. Following our intuition on the performance metrics used for an e-commerce website microservice application deployed in Kubernetes, the state space definitions of the research, and what we could actually measure with our tools, we believe that the set of metrics comprising *number of replicas*, *average pod CPU utilisation*, 95^{th} *percentile service latency*, *average pod memory utilisation*, and *average RPS received by the pod*, is sufficient to answer our research questions.

3.2 Problem formulation

For the RL agent to solve its task, we define the optimisation problem as follows.

Let $\mathcal{S} = \{1, \dots, S\}$ be the set of services in the considered microservice architecture. We assume that each pod contains a single service $s \in \mathcal{S}$. Then, we would like to minimise the number of concurrent pods for a service $N_s(t)$ at time $t \leq T$, where T is the time horizon subject to the service's latency constraint $\bar{L}_s$ corresponding to the SLO.

$$\begin{aligned} &\min \int_0^T \frac{N_s(t)}{T}\, dt \\ &s.t.\ \mathbb{E}[L_{s,t}] \leq \bar{L}_s,\ \forall s \in \mathcal{S},\ \forall t \leq T \end{aligned} \tag{3.1}$$

In addition, we assume that each pod is subject to a dynamic service request volume $w_s(t)$ where each successful response is served with a latency $L_{s,t}$. We assume that each pod has access to bounded resources such as CPU and memory, and that, depending on the service, it consumes a certain portion of these resources in order to process a response. The resource bound is given as input to the problem and it is assumed to be correctly set for each service s according to

the desired performance of a single pod subject to a request volume $w_s(t)$. Concerning the workload, we consider that each service receives an independent volume of requests $w_s(t)$, expressed in RPS, coming either from an external system or from another service within the microservice application. The traffic pattern is assumed to be dynamic in nature, including periodic patterns and randomness as sudden traffic spikes.

Furthermore, we consider that for each service S there is a single agent Ψ_s that controls the scaling action by augmenting, reducing, or keeping the number of pods constant for service s. The number of pods that can be scheduled is comprised within a minimum of 1 and a maximum of N_{max}, where N_{max} is provided as input to the problem. As input parameters for the scaling decision, the agent Ψ_s has access to $N_{s,t}$ and a set of pod performance indicators $\boldsymbol{S} = \{m_1, m_2, ..., m_n\}$ representing at each time instant t the mean value of each performance indicator $m \in \boldsymbol{S}$ computed across all active pods of service s. For example, some performance indicators that could be considered are the pod CPU utilisation and the request volume received by the pod.

Given this context, we want to find which set of performance metrics $\boldsymbol{S}$ inputted to the agent Ψ_s provides the best outcome for the above-mentioned optimisation problem, that is, minimise the concurrent number of pods N_s and the number of SLO violations.

3.2.0.1 States

We define our finite MDP problem as follows. Let $N_{s,t}$ be the number of replicas for service s at time t, let $C_{s,t}$ be the mean CPU utilisation of service s at time t, $M_{s,t}$ be the mean memory utilisation for service s at time t, let $L_{s,t}$ be the 95^{th} percentile service latency of service s at time t, let $H_{s,t}$ be the mean request rate per second for service s at time t. We define a set of state spaces $\Sigma = \{\boldsymbol{S}_1, \boldsymbol{S}_2, \dots \boldsymbol{S}_6\}$, where each set $\boldsymbol{S}$ is a different selected combination of the states

$$\boldsymbol{S} = \{C_{s,t}, H_{s,t}, L_{s,t}, M_{s,t}, N_{s,t}\}$$

The choice of these metrics follows our problem formulation which, by definition, includes $N_{s,t}$ and $L_{s,t}$, while the resource saturation metrics $C_{s,t}$ and $M_{s,t}$, and the application-level metric $H_{s,t}$ were chosen

according to the approaches analysed in the literature, in particular [36], [48], [50], [59], [70]. Specifically, the state space definitions tested are:

1. $\boldsymbol{S}_1 = \{C_{s,t}, H_{s,t}, L_{s,t}, M_{s,t}, N_{s,t}\}$, all metrics
2. $\boldsymbol{S}_2 = \{C_{s,t}, L_{s,t}, M_{s,t}, N_{s,t}\}$, without RPS
3. $\boldsymbol{S}_3 = \{C_{s,t}, L_{s,t}, N_{s,t}\}$, without RPS and memory
4. $\boldsymbol{S}_4 = \{L_{s,t}, M_{s,t}, N_{s,t}\}$, without CPU and RPS
5. $\boldsymbol{S}_5 = \{H_{s,t}, L_{s,t}, N_{s,t}\}$, without memory and CPU
6. $\boldsymbol{S}_6 = \{L_{s,t}, N_{s,t}\}$, only with latency and number of replicas

3.2.0.2 Actions

The discrete action space

$$\boldsymbol{A} = \{-1, 0, 1\}$$

is the set of possible actions $A_t \in \boldsymbol{A}$ that the agent can select at time t. Specifically, $A_t = -1$ corresponds to a *scale-in* action and reduces the number of replicas for the service s by 1. Action $A_t = +1$ corresponds to a *scale-out* action and increases the number of replicas for the service s by 1. Lastly, action $A_t = 0$ corresponds to the *do nothing* action and does not change the number of replicas for service s.

3.2.0.3 Rewards

The reward function is designed to penalise the violation on the service latency constraint $\bar{L}_s$, while minimising the number of concurrent pods $N_{s,t}$. Therefore, we define the reward function R_t at time t as

$$R_t = \begin{cases} -100 \times \tau & \tau > 0 \\ 100 \left(\dfrac{100}{\alpha} \times \tau + 1 \right) + 10 \dfrac{N_{MAX}}{N_{s,t}} & \tau \leq 0 \end{cases} \tag{3.2}$$

Where $\tau = \dfrac{L_{s,t}}{\bar{L}_s} - 1$ is the response time difference measured at time t with respect to the SLO, normalised with respect to $\bar{L}_s$. It represents the main constraint for the agent, and it is negative when the SLA

is violated, positive when it is met. The parameter $\alpha \in (0, 100]$ is a constant selected by the user which regulates the relative optimisation with respect to the SLA, similarly to the *excess* parameter in [59]. The intuition behind this is that $L_{s,t} \propto N_{s,t}^{-1}$, thus, with more replicas of service s the average response time $L_{s,t}$ decreases *. Therefore, the constant α gives the possibility to penalise the agent if the response time is far lower than the target SLO, which implicitly results in overprovisioning, by defining an optimisation range between 0% and 100%, e.g. $\alpha = 20$ would penalise the agent if $L_{s,t} < 0.2 \times \bar{L}_s$. Lastly, N_{MAX} is the maximum number of replicas the agent can use for the selected service s, while $N_{s,t}$ is the current number of replicas for service s.

The reward is maximum when $\tau \approx 0$ and $N_{MAX} \gg N_{s,t}$. Finally, the goal of the agent is to maximise the total discounted return $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ over time.

3.3 Experimental Setup

In this section, we discuss the details of our experimental setup, in particular, we consider the interactions between the components in Figure 3.3.1, a simplified representation of the implemented architecture †.

In the cloud platform, we deployed the Kubernetes cluster, which consists of a Master node and multiple worker nodes, which will host all the pods needed by the microservice application *Online Boutique*. Each pod contains one or multiple Docker containers, represented with a container with rounded angles in the Figure. For example, the container *RL agent* contains the code to use the model on the cluster when trained, while *Envoy* represents the sidecar proxy injected by Istio on every pod, next to the *Server* container. Each *Server* represents a microservice packaged into a container, for example, *productcatalog*, *frontend*, etc. Each pod is assigned to a node by the *kube-scheduler* based on resource availability, and the whole *Online Boutique* application runs in the same namespace.

*This is in accordance to our empirical results, see Section 3.7.1

†Note that, for simplicity, monitoring pods such as Prometheus and Grafana are not placed inside a node. In addition, pods may be assigned by Kubernetes to different nodes than the one represented in the Figure.

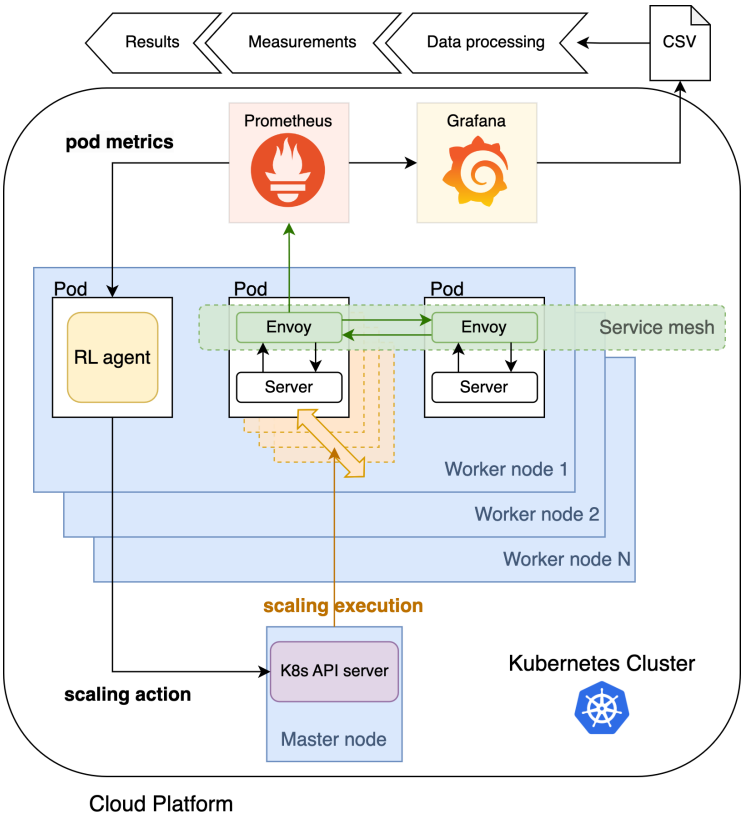


Figure 3.3.1: Simplified representation of the experimental architecture designed for this thesis.

The communication between microservices occurs through the service mesh enabled by Istio. Specifically, Envoy proxies monitor each server and publish these data on an endpoint. Both Prometheus and Grafana are deployed as pods inside the cluster. Prometheus scrapes every $5s$ the metrics from Istio, which collects these from each Envoy proxy present in the cluster. These metrics are stored in the Prometheus time-series database and are scraped by Grafana and the RL agent. With Grafana, we can visualise the rate of change of each metric over time in a dashboard, and download this data in our local development environment to process it and make our measurements.

The scaling action is performed by the RL agent container based on the observed state of the cluster. In fact, the agent queries Prometheus to receive aggregated pod metrics, representing state S_t, and takes a scaling action A_t accordingly, by calling the *kube-api-server* in the master node. Kubernetes is then responsible for executing the scaling action by increasing or decreasing the number of replicas in the target deployment.

With respect to the hardware and software tools used in our implementation, we report below our local development environment and cloud environment.

Local development environment

```
Computer hardware
Processor: 2.4GHz 8-Core Intel Core i9
Memory: 16GB 2667MHz DDR4

Main software tools
OS: macOS Ventura v13.4.1
python: version 3.11.1
pip: version 23.1
gym: version 0.21.0
stable-baselines3: version 1.7.0
tensorflow: version 2.12.0
torch: version 2.0.0
```

Cloud development environment

```
Google Kubernetes Engine (GKE)
Kubernetes cluster: version 1.25.10-gke.1200
number of nodes used: minimum 4, maximum 80

Node
Machine type: e2-standard-4
Image: container-Optimized OS with containerd
(cos_containerd)
vCPU allocatable: 3.92cores
vMemory allocatable: 13.93Gb

Istio Service Mesh
istioctl: version 1.13.3
Installation profile: default
Envoy proxy Docker image:
docker.io/istio/proxyv2:1.13.3
Prometheus Docker image: prom/prometheus:v2.34.0
Grafana Docker image: grafana/grafana:9.0.1

Microservice demo application
Source repository: v0.5.2 commit a14142d from [20]
Modified version at [8]
vCPU requested: 5070m
vMemory requested: 3772Mi
More details on resources at Table 3.3.1

Additional monitoring server
kube-state-metrics server: v.2.8.1 commit fb15826
installed from [39]
```

Deployment	Request (CPU/Memory)	Limit (CPU/Memory)	Sidecar Proxy (CPU/Memory)
adservice	$200m/180Mi$	$300m/300Mi$	$250m/256Mi$
cartservice	$200m/64Mi$	$300m/128Mi$	$250m/256Mi$
checkoutservice	$300m/64Mi$	$500m/128Mi$	$250m/256Mi$
currencyservice	$200m/64Mi$	$300m/128Mi$	$250m/256Mi$
emailservice	$100m/64Mi$	$200m/128Mi$	$100m/256Mi$
frontend	$300m/64Mi$	$400m/128Mi$	$500m/256Mi$
paymentservice	$100m/64Mi$	$200m/128Mi$	$250m/256Mi$
productcatalogservice	$250m/64Mi$	$300m/128Mi$	$500m/256Mi$
recommendationservice	$300m/64Mi$	$500m/128Mi$	$250m/256Mi$
redis-cart	$70m/200Mi$	$125m/256Mi$	$100m/256Mi$
shippingservice	$100m/64Mi$	$200m/128Mi$	$250m/256Mi$

Table 3.3.1: Resource Requests and Limits for microservice demo application.

3.4 Monitored metrics

In this section, we report the Prometheus queries we exploited to obtain the states $\boldsymbol{S}$ of the environment. The sampling frequency used by Prometheus is 5 seconds, and the rate of change of the metrics is averaged over the past $60s$, thus over the past 12 data samples. The latency introduced by the monitoring process in negligible considering that the update cycle of the agent is $30s$, while the Prometheus scarping time is in the order of $10^{-3}s$.

In the following we describe how each state is monitored, to provide an understanding of what the observation includes.

3.4.0.1 Number of replicas $N_{s,t}$

To count the number of active replicas, we exploit the metric `kube-pod-info` provided by the *kube-state-metrics* server. We count each active pod in the selected namespace with Metric 3.1, which is dimensionless.

```
count(
    kube_pod_info{
        pod=~"productcatalogservice-.*",
        namespace="<namespace>"}
```

```
    )
```

Metric 3.1: Replicas per deployment

3.4.0.2 Total load to deployment $\hat{H}_{s,t}$

Although this metric is not part of the agent's observation in our experiment, it has been used to simulate the real environment, see Section 3.7. The total load to the deployment is the rate of requests per second received by the deployment *productcatalog* in a single namespace over the past minute. This Metric 3.2 is expressed in req/s.

```
sum(
    rate(
    istio_requests_total{
        reporter="destination",
        namespace="<namespace>",
        destination_workload="productcatalogservice"
        }[1m]
        )
    )
```

Metric 3.2: Incoming load to the deployment

3.4.0.3 Average load to each replica $H_{s,t}$

Since the load is evenly distributed among the replicas in a deployment, we can obtain the average number of RPS destined to each replica *productcatalog*. We sum the arrival rate of requests received by each pod over the past minute, and take the mean to have the average RPS received by each pod. This Metric 3.3 is expressed in req/s.

```
avg(
    sum by (pod) (
        rate(
            istio_requests_total{
                reporter="destination",
                namespace="<namespace>",
                destination_workload="productcatalogservice"}
            [1m]
            )
```

```
        )
    )
```

Metric 3.3: Average RPS per pod in the deployment

3.4.0.4 Average CPU utilisation for each replica $C_{s,t}$

Each pod consumes a quantity of vCPU from the host machine, and we compute the ratio between the consumed CPU and the requested CPU. For each pod in the deployment, we divide the cores used each second by the core requested, and average across all pods to have a quantity representative of the vCPU utilisation of each pod. This Metric 3.4 is dimensionless.

```
avg(
    sum by (pod) (
    rate(
        container_cpu_usage_seconds_total{
            namespace="<namespace>",
            pod=~"product.*",
            image!=""
        }
        [1m]
        )
    )
    /
    sum by (pod) (
        kube_pod_container_resource_requests{
            namespace="<namespace>",
            unit="core",
            pod=~"product.*"
            }
        )
    * 100)
```

Metric 3.4: Average CPU per pod

3.4.0.5 Average memory utilisation for each replica $M_{s,t}$

The average vMemory utilisation for each pod is computed similarly to vCPU utilisation. The memory in Metric 3.5 is dimensionless.

```
avg(
    sum by (pod) (
    rate(
        container_memory_usage_bytes{
            namespace="<namespace>",
            pod=~"product.*"}
        [1m])
    )
    /
    sum by (pod) (
        kube_pod_container_resource_requests{
            namespace="<namespace>",
            unit="byte",
            pod=~"product.*"}
        )
    * 100)
```

Metric 3.5: Average memory per pod

3.4.0.6 95^{th} percentile latency for each replica $L_{s,t}$

To measure the average response time, we consider the 95^{th} percentile tail latency, corresponding to the slowest 5% of the requests served by the pod. Specifically, the service latency $L_{s,t}$ is measured by Envoy proxy as the time elapsed from when the Envoy proxy receives the request from an external service, to when the request is sent back to the client service. We only consider the latency of successful requests with HTTP Status Code 200, as other codes representing unsuccessful requests may still have optimal latency but are not useful to the client. In Metric 3.6, we use the Prometheus operator `histogram_quantile()` on the histogram metric `istio_request_duration_milliseconds_bucket` to compute the durantion in milliseconds of successful requests.

```
histogram_quantile(
    0.95,
    sum by (le) (
        rate(
            istio_request_duration_milliseconds_bucket{
                reporter="destination",
                destination_service_namespace="<namespace>",
                response_code="200",
```

```
                pod=~"productcatalogservice-.*"}
            [1m]
        )
    )
)
```

Metric 3.6: 95^{th} percentile tail latency of HTTP 200 requests

3.4.0.7 Average CPU utilisation by image in the application

Although Metric 3.7 is not part of the agent's observation, it has been used to compute the minimal number of replicas for each deployment in the application that could withstand the highest load generated by *loadgenerator*.

```
sum by (image)
(label_replace(
    rate(container_cpu_usage_seconds_total{
        namespace="<namespace>", image!=""}
        [1m]
        ),
        "image", "$1", "pod", "^([^\\-]+).*"
        )
)
/
sum by (image)
(label_replace(
    kube_pod_container_resource_requests{
        namespace="<namespace>", unit="core"},
        "image", "$1", "pod", "^([^\\-]+).*"
        )
) * 100
```

Metric 3.7: Average CPU utilisation by image

3.5 Planned measurements and modelling

To answer our research questions, we primarily intend to measure the total reward G_T obtained during the learning phase when the A2C agent is given different observation spaces. Then, once the learning

is completed, we assess the performance of the trained model in terms of SLA violations and number of pod replicas used. To answer our research questions, we design two main experiments. The first, aims at studying how different environment observations affect the performance of the agent on various workload patterns. The second, aims at comparing the Kubernetes HPA baseline autoscaler with the model-free A2C agent, on the same workload patterns. The detailed procedure we adopted for each experiment is reported in Chapter 4, followed by the results and analysis in Chapter 5.

We model our problem as a real microservice application deployed in a real Kubernetes cluster in the cloud. However, to perform our measurements, we have to consider the following. Firstly, we cannot generate real traffic to the application coming from real users because the application could not be deployed in a production environment. For this reason, we consider an *emulation* of the target system in which traffic workloads are synthetically generated with a load generator simulating the behaviour of real users. This approach has been taken in other studies as well [69]. Secondly, we measured the emulated environment with the aim of creating a simple but effective *simulation* that could be used to train the agent. We registered the behaviour of the application in the real GKE cluster, and used this data to replicate the behaviour in our local development environment. Our motivations for this latter design choice are discussed in Section 3.5.1. We then deploy our agent, trained in the simulated environment, on the emulation and compare it with the HPA.

3.5.0.1 Selecting a target backend service

Regarding our design implementation, we opted for a fully decentralised solution, which, although there are cases in which it performs worse compared to hierarchical solutions [50], is simple enough to allow us to perform our measurements and draw valid conclusions. Therefore, in our architecture, there is a one-to-one correspondence between microservices and agents, which are all deployed in the same namespace.

We selected a representative backend service, namely *productcatalog*, on which to base our measurements as we did not have resources to measure all eleven microservices. Despite this, we consider

the target microservice a representative sample for a general cloud-based microservice because of its characteristics. In particular, *productcatalog* is frequently called by *frontend* and many other microservices, making it the ideal candidate as it serves requests from many sources. Moreover, we avoided the selection of *frontend* as target because, as shown in [69], the performance degradation measured in such service may not be related to *frontend* itself, rather, to other congested backed services. Finally, the methodology designed for this service can be applied to any other target microservice.

3.5.0.2 Ensuring statistical significance

We try to make our measurements statistically significant by performing multiple trials and taking the mean and standard deviation. Although the cost of a measurement is high in terms of time and resources, we limited it to a few numbers of trials, which nonetheless proved to be statistically relevant.

For example, when it comes to gathering data to simulate the Kubernetes system, we run seven *Online Boutique* applications in parallel in seven different namespaces and aggregate the results to compute the mean μ and standard deviation σ of our metrics. In addition, when training the Actor-Critic model, we perform ten trials and aggregate the data on the total accumulated reward G_T.

3.5.1 Simulation

By design choice, we begin by running a simulation of the environment and perform trainings in our local setup and not directly in GKE. The main motivation to run a simulation is to avoid the long training time that would otherwise be needed in the real environment. In fact, after a scaling action is taken by the agent, to compute the reward based on the new state of the environment, it is necessary to wait a period corresponding to the container startup time and the time needed for the load balancer to distribute the load among new replicas [70].

This time interval ranges between 10 and 30 seconds for the *productcatalogservice* pod * , therefore, pausing the agent before

*To measure the container startup time we used
`kube_pod_status_ready_time{namespace="<namespace>", pod= "product.*"}`

computing the reward and taking the next scaling action would incur in a training time of $t \times \Upsilon$, where t is the time required for the replica to be ready, Υ the number of actions that the agent must take in the time horizon T. Since the number of steps is usually in the order of at least 10^5, a single training could take a minimum of $30s \times 10^5 \approx 830hours$. On the contrary, in a simulated environment the resulting state from a scaling action is hard-coded into the system, and we can give immediate reward to the agent, thus speeding up the learning phase to just a few minutes with our hardware. Moreover, the simulation allows us to learn faster from the agent's behaviour with a series of iterative tuning cycles that would not be possible with the agent deployed in the actual Kubernetes cluster.

Therefore, to create a simulation model, we mainly have to simulate the relationship between $N_{s,t}$ and ($C_{s,t}$, $H_{s,t}$, $L_{s,t}$, $M_{s,t}$), and simulate the reward R_t. This is explained in details in Section 3.7.

3.6 Backend microservice performance evaluation

The scope of this experiment is to measure the 95^{th} percentile (p_{95}) average response latency of a pod under normal and overloaded conditions for the service *productcatalog*. This value will be used to set the SLO $\bar{L}_s$ that the RL agent will learn to comply with, similarly to what [70] have proposed.

We do so by increasing the load to one single replica of the deployment, and register the RPS, the p_{95} latency, the average CPU utilisation with respect to the requested CPU, and the memory utilisation with respect to the requested memory. Moreover, to have statistically relevant results, we perform the same experiment in 7 different namespaces in the cluster, and we consider the mean μ and standard deviation σ of these measurements.

With our measurements, we want to generate a table where for each number of simulated users generating traffic on the application, we register the state of the pod. What we expect to see is that the p_{95} latency

-

```
kube_pod_created{namespace="<namespace>", pod= "product.*"}
```

will remain at a stable level with increasing load until a point when the load will be so high that the latency will start to increase, thus resulting in the abnormal behaviour.

3.6.0.1 Methodology

Since the scope of this measurement is to assess the behaviour of the backend service *productcatalog*, it is important to isolate its performance from the influence of other services in the application, as [69] have noted. In fact, since *productcatalog* is called by *frontend* when a simulated client generated by *loadgenerator* uses the application, we make sure that *frontend* does not become the bottleneck and prevent requests to arrive to our target backed, by increasing its replicas. In addition, also other services, such as *recommendation*, *ad*, *checkout* and *currency*, are called when users interact with *frontend*, therefore, we increase the number of replicas for these microservices too. Specifically, we measured with Metric 3.7 the average CPU utilisation of those services and made sure that with the given load from 2000 users, the value was always below 100%. To remove possible bottlenecks, we concluded that 6 replicas had to be given to the other deployments.

Our goal is to measure $C_{s,t}$, $H_{s,t}$, $M_{s,t}$, $N_{s,t}$, $L_{s,t}$, and obtain $\bar{L}_s$ by increasing the workload to a single replica of *productcatalog*. Therefore, in each namespace, with *loadgenerator*, we generate a step-wise workload, each step of a duration of 5 minutes. Starting with 1 user, at each subsequent step we add 50 users, until reaching a total of 500 clients using the application. We stopped at 500 since it was possible to record $\bar{L}_s$ before that.

To compute the mean μ and standard deviation σ we proceeded as follows. In each namespace n, for each interval in which the number of users is kept constant, i.e. $\Delta t = t_{end} - t_{init} = 5'$, we computed the mean value $\bar{\phi}_{n,i}$ of the signal ϕ in $i = [t_{end} - \frac{\Delta t}{2}, t_{end}]$, which is a time window of 2.5 minutes, and ϕ being the metric signal measured over time, i.e. $H_{s,t}$, $L_{s,t}$, $C_{s,t}$, and $M_{s,t}$. This is because we wanted the system to settle down after the number of users is increased abruptly, see Figure 6.1.1 in the Appendix. Then, for each interval i, we computed the mean value μ and standard deviation σ for each metric $\bar{\phi}_{n,i}$ across all namespaces.

With this procedure, we were able to identify the performance of

productcatalog under normal and stressful conditions, in particular, to identify the target latency $\bar{L}_s$, our SLO. The results of these measurements are reported in Section 3.6.1.

3.6.1 Performance results for a single replica

In this experiment, the goal was to measure when the p_{95} latency deviates from the normal value by increasing the load on a single replica of *productcatalog*. As reported in Table 3.6.1, we can see that $L_{s,t}$ is consistently below $5ms$ up to a load $H_{s,t}$ of about $500req/s$. Then, once this threshold is overcome, $L_{s,t}$ rapidly increases. For this reason, we define $\bar{L}_s = 5ms$ as SLO, and consider a single replica of *productcatalog* overloaded when $L_{s,t} > \bar{L}_s$.

In addition, we note that *productcatalog* is a CPU-intensive service. In fact, as the workload $H_{s,t}$ increases, the CPU utilisation $C_{s,t}$ increases with the same trend, but the memory utilisation $M_{s,t}$ remains on a constant level.

Users	$H_{s,t}$ [req/s]		$L_{s,t}$ [ms]		$C_{s,t}$ [%]		$M_{s,t}$ [%]	
	μ	σ	μ	σ	μ	σ	μ	σ
1	1.38	0.27	4.42	0.28	1.97	0.12	0.78	0.14
50	71.77	1.94	4.35	0.31	16.57	1.37	0.71	0.18
100	143.34	3.6	4.25	0.35	29.27	2.27	0.79	0.15
150	215.71	2.99	4.14	0.4	40.65	3.4	0.63	0.2
200	288.13	2.93	4.21	0.39	53.04	4.81	0.73	0.21
250	359.4	2.5	4.15	0.56	62.45	5.84	0.71	0.14
300	429.21	4.49	4.25	0.46	70.92	5.17	0.76	0.24
350	**504.03**	8.06	**4.65**	0.5	**81.71**	8.15	**0.8**	0.28
400	570.37	5.26	7.53	3.7	86.4	7.3	0.75	0.24
450	646.43	5.12	11.05	7.56	92.2	7.73	0.7	0.25
500	714.63	4.35	15.53	10.86	99.12	6.1	0.78	0.2

Table 3.6.1: One replica of *productcatalog* with increasing load. Highlighted the maximum load before performance degradation.

3.7 Data collection for the environment simulation

The purpose of this measurement is two-fold. On one side, we want to take data samples from the environment emulation that would enable us to approximate the states of the system used by the RL model. Secondly, we want to validate the hypothesis that a scaling-out action can mitigate the SLA violation when the service is overloaded.

In our simulation of the environment, we do not have a real Kubernetes cluster on which the agent is deployed, nor an infrastructure to measure the performance of a pod under different states. Therefore, we need to hard-code the sequence of MDP steps that the agent would take in the real environment. Specifically, the RL agent needs an action signal A_t, a state signal S_t, and a reward signal R_{t+1}. Once these signals are provided to the model, the training can be completed in the local development environment.

In particular, we simulate the sequence $S_t \rightarrow A_t \rightarrow R_{t+1} \rightarrow S_{t+1}$ by measuring how the real system transitions from state S_t to S_{t+1} in every state-action pair scenario. We generate a table where the agent can query the state $\boldsymbol{S}_n$ given the current number of replicas $N_{s,t}$ and the workload to the deployment $\hat{H}_{s,t}$ *, expressed in RPS.

3.7.0.1 Methodology

To generate samples that would provide the data for the relationship

$$(C_{s,t}, H_{s,t}, L_{s,t}, M_{s,t}) \leftarrow (N_{s,t}, \hat{H}_{s,t})$$

we generated a workload $\hat{H}_{s,t}$ from $0req/s$ to $2000req/s$ for each number of *productcatalog* replicas, from $N_{min} = 1$ to $N_{MAX} = 15$.

Once again, we deployed 7 parallel versions of the *Online Boutique* web service in 7 namespaces in the Kubernetes cluster, increased the number of replicas of other services related to *productcatalog* to 6, and used *loadgenerator* to generate traffic by simulating real user

*Note that we express $H_{s,t}$ as the average RPS to each pod in the deployment, while $\hat{H}_{s,t}$ is the absolute RPS coming to the deployment, which is then balanced among pods.

Sample	$N_{s,t}$	$\hat{H}_{s,t}$	$C_{s,t}$	$H_{s,t}$	$L_{s,t}$	$M_{s,t}$
x_1	2	540.29	36.64	270.0	4.0	0.44
x_2	2	574.0	39.73	286.6	4.03	0.47

Table 3.7.1: Example of two collected samples.

behaviour. Then, for each metric signal we averaged the values across namespaces.

Algorithm 1 Generate increasing workload for each $N_{s,t} \in [1, 15]$

$N_{s,t} \leftarrow 1$
$\hat{H}_{s,t} \leftarrow 0$
while $N_{s,t} <= 15$ **do**
  while $\hat{H}_{s,t} < 2000$ **do**
    Record $(C_{s,t}, H_{s,t}, L_{s,t}, M_{s,t}, N_{s,t})$ from the deployment *productcatalog*
    Wait $20s$ for the system to stabilise
    $\hat{H}_{s,t} \leftarrow \hat{H}_{s,t} + 70$
  $N_{s,t} \leftarrow N_{s,t} + 1$
  $\hat{H}_{s,t} \leftarrow 0$

As described in Algorithm 1, we increase the workload by $70req/s$, corresponding to 50 simulated users, until we reach $2000req/s$ with $N_{s,t}$ fixed. We repeat the process by increasing the number of replicas until we reach 15. The results are reported in the following Section 3.7.1.

Due to the fact that with *loadgenerator* we control only the number of users and not directly the RPS generated, we do not have a continuous set of samples, as to 1 user does not correspond exactly $1req/s$. In addition, to avoid a very lengthy measurement, we ramped up users by sets of 50 at each step. For example, two successive data points are x_1 and x_2 in Table 3.7.1. Though, when we train the RL agent on a custom workload, we may have the state ($N_{s,t} = 2$, $\hat{H}_{s,t} = 550$), which is not covered in the table. For this reason, we need to approximate the values of observations between data samples.

We apply the function $f(x_n)$ in Equation 3.3 to approximate metric values between samples, by taking the relative distance with respect to x_n between data points x_{n-1} and x_{n+1}.

$$\beta = \left| \frac{x_n - x_{n-1}}{x_{n+1} - x_{n-1}} \right| \tag{3.3}$$
$$f(x_n) = x_{n-1} + \beta(x_{n+1} - x_{n-1})$$

However, two limit cases occur when $\hat{H}_{s,t}$ is lower than the minimum $\hat{H}_{min}$ or higher than the maximum $\hat{H}_{MAX}$, being the lowest and highest sample values recorded in the measurement, respectively.

In the case in which $\hat{H}_{s,t} < \hat{H}_{min}$, we perform a linear regression on the smallest 20 values of each metric ϕ to obtain the slope m and intercept q for the linear function $\phi(\hat{H}_{s,t}) = m \times \hat{H}_{s,t} + q$ to approximate the metric value for the unknown sample.

In the case in which $\hat{H}_{s,t} > \hat{H}_{MAX}$, we adopt the same linear regression strategy as in the previous case, with the largest 20 values.

Lastly, we believe that our approximations are valid to train the agent on a simulation, and that its learned policy can be applied in a real cluster, as we will show in Chapter 5.

3.7.1 Performance results for multiple replicas

In Table 3.7.2, we report a representative sample for the values we obtained in our measurements. Specifically, we report for a constant load $\hat{H}_{s,t} = 1500req/s$ the values of each state space, along with the expected reward from Equation 3.2. While in Figure 3.7.1, the plots for each metric in our measurements following Algorithm 1. Plotted in blue the absolute values of each metric ϕ from each namespace, in red the mean of these values, which is used to generate the query table. As it is possible to see, while number of replicas is kept constant, the load to the deployment increases linearly. During this interval, we measured our states. We can now train the model using these data, as we have modelled our emulated environment.

Lastly, from Table 3.7.2 we can see that in fact a scaling-out action mitigates the SLA violation, since increasing the number of replicas distributes the load to the deployment among its replicas. Specifically, for $\hat{H}_{s,t} = 1500req/s$, the optimal number of replicas is 3 since it is the minimum number $N_{s,t}$ that provides a performance under $\bar{L}_{s,t} = 5ms$.

$N_{s,t}$	$H_{s,t}$ [req/s]	$L_{s,t}$ [ms]	$C_{s,t}$ [%]	$M_{s,t}$ [%]	R_t
1	1500.0	70.59	123.04	0.44	−1311.82
2	749.93	6.88	94.61	0.55	−37.57
3	**500.08**	**4.34**	**72.35**	**0.51**	**62.4**
4	374.88	4.12	57.43	0.45	20.5
5	300.03	4.17	48.88	0.5	18.9
6	250.07	4.12	41.88	0.49	7.7
7	214.32	4.24	37.37	0.48	20.1
8	187.5	4.31	34.11	0.5	26.8
9	166.54	4.38	30.92	0.49	34
10	149.95	4.24	27.47	0.46	13.7
11	136.56	4.33	25.56	0.48	20.4
12	125.03	4.36	23.84	0.52	2.72
13	115.42	4.25	21.82	0.5	1.15
14	107.0	4.34	20.68	0.5	2.24
15	100.01	4.3	19.48	0.48	1.62

Table 3.7.2: Constant load $\hat{H}_{s,t} = 1500 req/s$ to *productcatalog* for each $N_{s,t}$.

Moreover, this number is correctly identified by the designed reward function, which provides the highest reward for that replica. Note that $N_{s,t} > 3$ provides acceptable performance too, but the higher number of replicas needed is penalised by the function.

3.8 Learning algorithm

We deploy a model-free Actor-Critic DRL agent in a Kubernetes cluster to perform scaling actions. The selection of the algorithm has been taken considering the current state-of-the-art in autonomous container orchestration presented in Section 2.6. Moreover, considering related work in the field, we identified the research gap introduced in Section 1.1, and we studied how different state space definitions Σ affect the performance of our model.

We decided to use A2C because it is a stable, relatively light implementation of an Actor-Critic DRL algorithm [37] and provides good performance when trained on CPU [33], which is our main processing hardware. In fact, we believe that the results obtained with

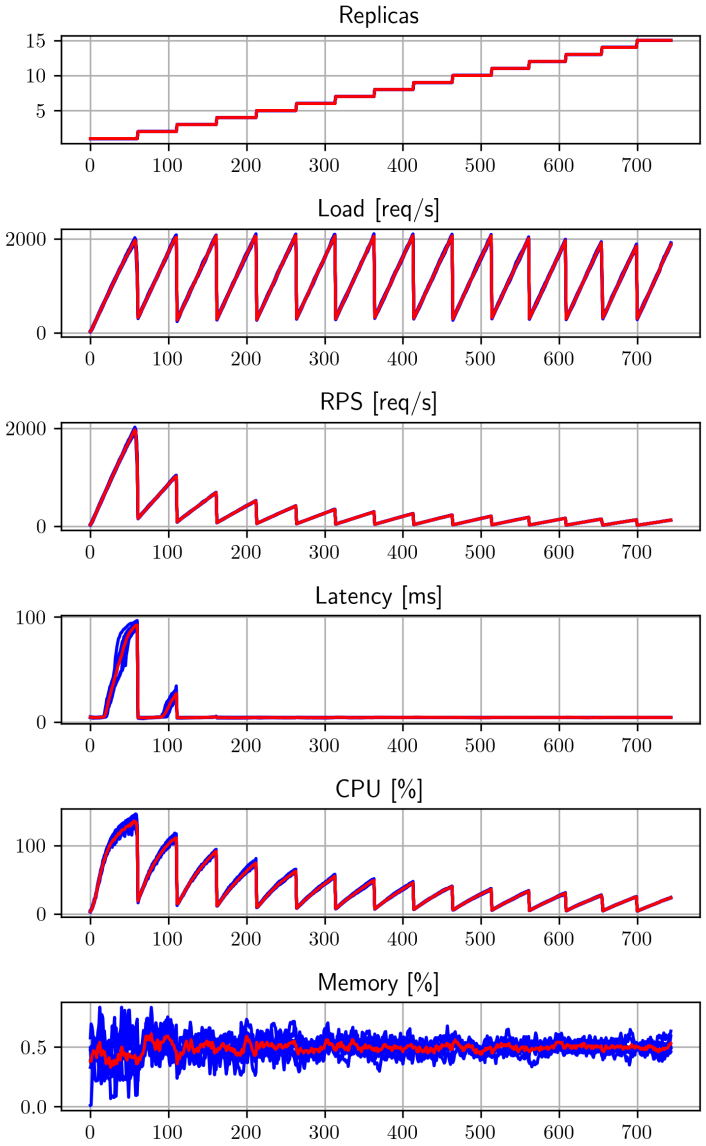


Figure 3.7.1: Mean signals used to generate data samples. Red signal indicates the mean over the 7 measurements, represented by the blue lines.

our implementation can be translated to other model-free Actor-Critic algorithms, thus this design choice will not affect the generality of our study.

The algorithm employed to train our A2C agent is Algorithm 2. Essentially, we exploit the Kubernetes API, Prometheus API, and the gym API [9] from OpenAI, to implement the scaling action, to retrieve the state of the environment, and to take actions and train the agent, respectively. To ensure environment exploration and avoid that the model occasionally remains stuck in a local-optima, every 1000 steps we enforce the choice of a random action selected among scale-out and scale-in.

After performing various experiments with different hyperparameters for the A2C implementation, we recorded acceptable performance with the parameters listed in Table 3.8.1. These parameters will be used by all our agents during training.

Another crucial aspect of DRL is the choice of the training length. Considering that our control task is in the infinite time horizon, meaning that the learning is not episodic, we need to select a finite number of steps T, which must be sufficiently high for the model to learn by exploring enough states of the environment. However, we need to terminate the training at time T to save the model and use it for prediction in our experiments. We select $T = 403020$ which corresponds to two weeks of training since the time interval between two steps is $30s$, for the reasons outlined in Section 3.5.1. In fact, from our measurements, the selected T is sufficient to observe an increasing trend in G_T, corresponding to a good learning behaviour.

For what concerns the performance overhead introduced by the use of the agent in the cluster, the resources consumed during the inference phase are $6m$ cores of CPU and $320Mi$ of memory, thus affecting the cluster performance with a minimal impact.

n_steps	learning_rate	gamma	gae_lambda	ent_coef	vf_coef
2	0.0014	0.95	1.0	0.5	0.5

Table 3.8.1: Learning hyperparameters.

Algorithm 2 Agent training in Kubernetes environment

function UpdateReplicas(A_t) ▷ Call to K8s API
    Output: N_{t+1}
    $N_t \leftarrow$ number of replicas for deployment d in namespace n from K8s API
    if $N_t + A_t > N_{MAX}$ **then**
        $N_{t+1} \leftarrow N_{MAX}$
    else if $N_t + A_t < N_{min}$ **then**
        $N_{t+1} \leftarrow N_{min}$
    else
        $N_{t+1} \leftarrow N_t + A_t$

function PromQuery(m) ▷ Call to Prometheus API
    Output: v
    $v \leftarrow$ query result for metric m from Prometheus API

function GetObservation($\boldsymbol{S}_n$)
    Output: S_t
    for metric m in $\boldsymbol{S}_n$ **do**
        $S_{t,m} \leftarrow$ PromQuery(m)

function Step(`model`, $\boldsymbol{S}_n$) ▷ Inherited from Stable Baselines 3
    Output: A_t, S_t, R_{t+1}
    $S_t \leftarrow$ GetObservation($\boldsymbol{S}_n$)
    $A_t \leftarrow$ `model`(S_t)
    if step is multiple of 1000 **then**
        $A(t) \leftarrow$ random choice between $[-1, +1]$
    UpdateReplicas(A_t)
    $R_{t+1} \leftarrow f(S_t, A_t)$

`model` $\leftarrow$ Actor-Critic (hyperparameters)
State space definition $\boldsymbol{S}_n \leftarrow$ selected from $\Sigma = \{\boldsymbol{S}_1, \boldsymbol{S}_2, ..., \boldsymbol{S}_6\}$
while training is not done **do**
    A_t, S_t, $R_{t+1} \leftarrow$ Step(`model`, $\boldsymbol{S}_n$)
    `model` $\leftarrow$ ANN weights update $\leftarrow A_t$, S_t, R_{t+1}
    $t \leftarrow t + 1$

Chapter 4

Experiments

In this chapter we present the two main experiments aimed at answering our research questions. In the first, we assess the influence of different observation sets on the model-free algorithm's ability to take optimal scaling actions, thus providing data to answer our **RQ1**. In the second experiment, we compare the best performers against the default Kubernetes HPA set on average CPU utilisation, with the goal of answering our **RQ2** and **RQ3**. In the following, we outline the steps needed to reproduce the experiments.

4.1 Simulation on different workload patterns

This first experiment consists in studying the performance of the agent trained with different state space definitions from the set Σ. As discussed in Chapter 3, we simulate the system behaviour in our local development environment and train the model by simulating the system's state changes given a certain action.

We consider the agent on 6 state space definitions, and evaluate each definition on 4 different workload shapes. The reason behind the choice of the workloads is to assess the performance under different scenarios, one of which resembles a real-world patten. We begin by testing on a constant load, then on an increasing step-wise load, on a periodic pattern, and on the same periodic pattern with random traffic spikes.

4.1.1 Procedure

The experiment consists of three main steps: the creation of a workload pattern in RPS as a function of time, the learning phase for the agent, and the inference phase, i.e. when we use the trained model to evaluate its performance in terms of SLA violations. In particular, during the inference phase for the two periodic signals, we validate the trained agent on slight variations of those patterns to assess the generality of the learned policy, as will be explained in the following.

4.1.1.1 Pattern generation

As described in Section 3.7, from the data samples gathered from the emulated cloud environment we can now approximate the states of the system with Equation 3.3, given the load $\hat{H}_{s,t}$ and the number of replicas $N_{s,t}$.

Therefore, we generate a discrete signal $\hat{\mathbb{H}}(t)$ expressed in req/s, where at each timestep $t \in [0, T]$ corresponds a load value to the deployment. In this way, at each subsequent timestep t, the agent will query for $(C_{s,t}, H_{s,t}, L_{s,t}, M_{s,t}) \leftarrow (N_{s,t}, \hat{\mathbb{H}}(t))$, thus obtaining the desired observation. We designed the following patterns:

- Constant workload
$$\hat{\mathbb{H}}_c(t, H) = H$$
- Stepwise workload
$$\hat{\mathbb{H}}_s(t, H_{min}, H_{MAX}, s, T) = H_{min} + \frac{(H_{MAX} - H_{min})}{s} \left\lfloor \frac{t \times s}{T} \right\rfloor$$
- Periodic workload
$$\hat{\mathbb{H}}_p(t, s, T, H_{min}, H_{MAX}) = \lambda + (H_{MAX} - \lambda) \times sin\left(\frac{2\pi}{T} st\right)$$
- Periodic workload with random spikes
$$\hat{\mathbb{H}}_r(t, s, T, \delta, H_{min}, H_{MAX}) = \delta(t) + \hat{\mathbb{H}}'_p$$

4.1.1.2 Measurements

For each state space definition, we performed 10 training trials with the agent on the same workload pattern, and averaged the results to ensure statistical significance.

We divided the experiment in a training phase and an inference phase. In the first phase, for each training trial, we computed the total accumulated reward G_T. Then, we took the mean μ_{G_T} and standard deviation σ_{G_T} among the trials as reference result. At the end of each training the models are saved and stored locally. In the inference phase, we loaded the best performing model for each state definition and performed a single trial in which we recorded the total accumulated reward G_T, the average number of replicas used $\mu_{N_{s,t}}$, its standard deviation $\sigma_{N_{s,t}}$, and the percentage of SLA violations. As SLA violation, we consider the number of times $L_{s,t} > \bar{L}_s$ divided by the number of steps T. In the inference phase, we perform a single trial for each state space since in a realistic scenario one would deploy the best model and exploit that for the whole length of the control task.

During the training phase, we consider that the model is learning a good policy if we register an increasing trend in G_t over time. In fact, as the agent steps into the simulated environment, it should accumulate positive rewards R_{t+1}. On the contrary, a negative reward implies an SLA violation resulting from a wrong action for that state. In the inference phase, the performance of the model is considered acceptable when the total accumulate reward G_T is high, and SLA violations and $\mu_{N_{s,t}}$ are minimised.

All models are trained with the hyperparameters reported in Table 3.8.1 and both learning and inference phases consist of the same number of steps T.

4.1.2 Constant load

The purpose of this experiment on constant load is mainly to validate the assumption that the model-free A2C agent can find an optimal policy for the formulated problem, given a constant number of users visiting the *Online Boutique* website. In fact, a constant load is unrealistic if we consider a real-world scenario in which some users interact with the web application. Nonetheless, this experiment will be

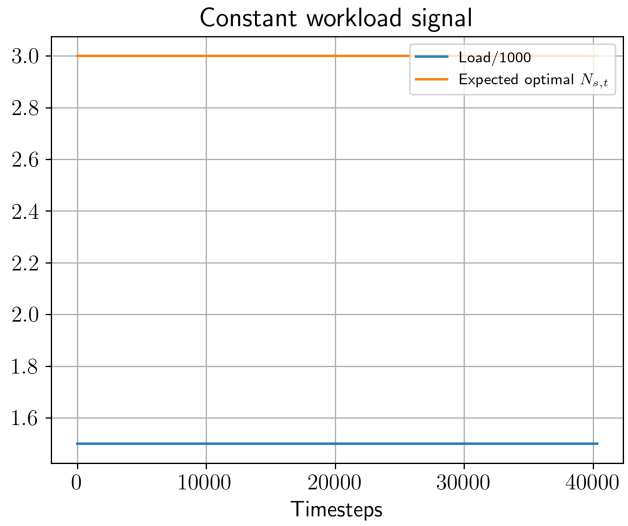


Figure 4.1.1: Constant load

primarily used to validate our implementation design, and allow us to make our first considerations on the system's behaviour.

The signal used is $\hat{\mathbb{H}}_c(t, H) = H$, where $H = 1500 req/s$. In Figure 4.1.1, we reported the load divided by 1000, to be comparable with the optimal number of replicas $N_{s,t}$ on the plot. With this workload, we expect the agent to learn that the optimal number of replicas is 3.

4.1.3 Stepwise load

Once the agent is able to learn the optimal policy for the constant workload $\hat{\mathbb{H}}_c(t)$, we further validate its efficacy with a more dynamic load.

As visible in Figure 4.1.2, the load implies a change in the optimal number of replicas needed. In particular, the stepwise workload used is

$$\hat{\mathbb{H}}_s(t, H_{min}, H_{MAX}, s, T) = H_{min} + \frac{(H_{MAX} - H_{min})}{s} \left\lfloor \frac{t \times s}{T} \right\rfloor$$

where $H_{min} = 150 req/s$ is the minimum number of RPS, $H_{MAX} = 4000 req/s$ is the maximum, and $s = 8$ is the number of steps.

In this experiment, we expect to see that different state space

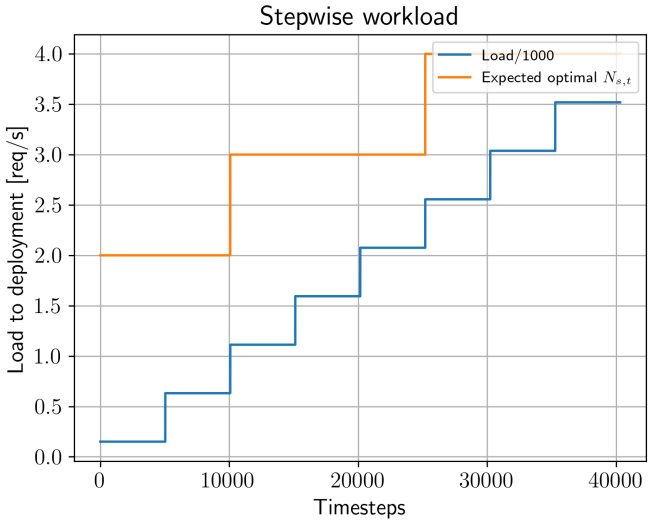


Figure 4.1.2: Step load

definitions provide different results, as some metrics may be more relevant than others for the model to learn the optimal policy in the considered time horizon. In Figure 4.1.2 we report the stepwise workload, along with the optimal number of replicas needed. The target mean replicas is $\mu_{N_{s,t}} = 3.12$.

4.1.4 Periodic load

The rationale behind the choice of the third pattern is to resemble a real-world workload in which there are peak traffic times during the day, as well as down-times in which fewer users visit the website. This pattern is repeated during the two weeks period, with one peak and one valley per day, ideally representing day and night hours.

The signal used is represented in Figure 4.1.3a over one period, where the optimal number of replicas oscillates between a maximum of 4 and a minimum of 1, while in Figure 4.1.3b the signal for the whole training length. The signal is defined as

$$\hat{\mathbb{H}}_p(t, s, T, H_{min}, H_{MAX}) = \lambda + (H_{MAX} - \lambda) \times sin\left(\frac{2\pi}{T} st\right)$$

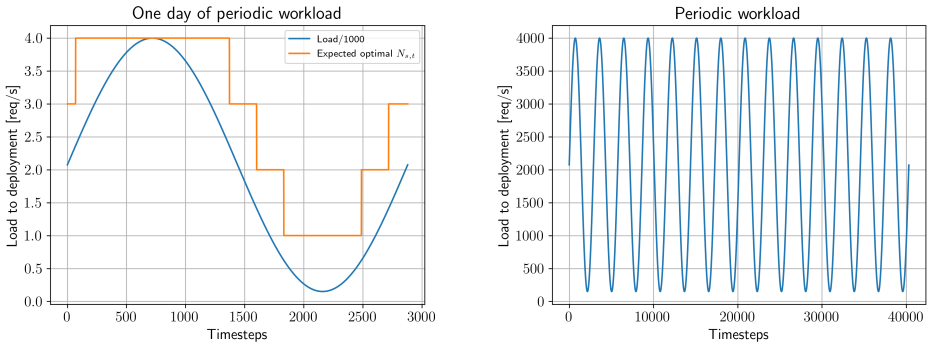
where $\lambda = \dfrac{H_{MAX} + H_{min}}{2}$. The parameters involved are $s = 14$ as

the number of periods, $H_{min} = 150req/s$ as the minimum load, and $H_{MAX} = 4000req/s$ as the peak load. The value λ is used to set the mean value of the load between the highest and lowest loads. The target mean replicas is $\mu_{N_{s,t}} = 2.84$.

During the inference phase, we will test the trained agent also on a variation of the periodic workload $\hat{\mathbb{H}}_p$ defined as

$$\hat{\mathbb{H}}_{pc}(t, s, T, H_{min}, H_{MAX}) = \lambda + (H_{MAX} - \lambda) \times cos\left(\frac{2\pi}{T} st\right)$$

with the same parameters as $\hat{\mathbb{H}}_p$.



(a) One period of periodic workload with optimal replicas.

(b) Periodic workload over two simulated weeks.

Figure 4.1.3: Periodic signal.

4.1.5 Periodic load with random spikes

Lastly, we introduce a periodic pattern with random increments and decrements to the load. This is to assess the ability of the agent to adapt to unpredictable patterns. Moreover, this scenario can be considered more realistic and closer to reality as sudden changes in the client demand are to be expected.

We modify the signal $\hat{\mathbb{H}}_p(t)$ by adding a signal $\delta(t)$ corresponding to a load value randomly chosen from a uniform distribution

$$\rho = \left[-\frac{H_{MAX} - H_{min}}{4}, +\frac{H_{MAX} - H_{min}}{4}\right]$$

with a duration randomly sampled from the uniform distribution $\rho' = [5, 20]$, between 5 to 20 timesteps. The spike probability is set to $p = 0.01$.

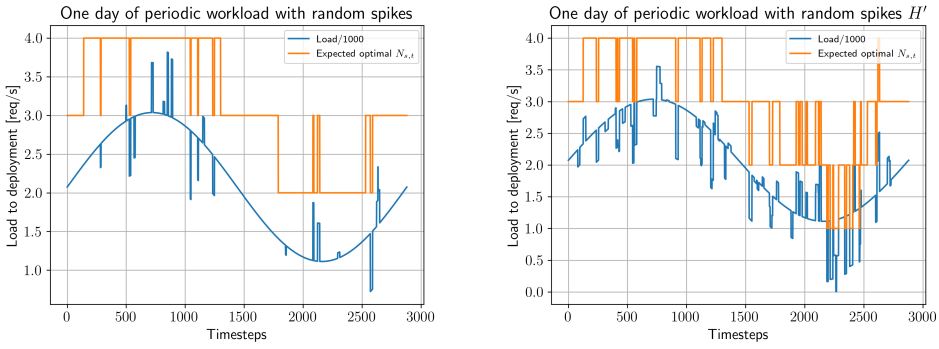
The resulting signal is

$$\hat{\mathbb{H}}_r(t, s, T, \delta, H_{min}, H_{MAX}) = \delta(t) + \lambda + \frac{1}{2}(H_{MAX} - \lambda) \times sin\left(\frac{2\pi}{T}st\right)$$

where $s = 14$ is the period, $H_{min} = 150req/s$, $H_{MAX} = 4000req/s$, and λ corresponds to the same parameter used for the periodic signal $\hat{\mathbb{H}}_p(t)$.

The first period of the function $\hat{\mathbb{H}}_r$ is visible in Figure 4.1.4a, together with the optimal number of replicas needed, while the whole length of the signal is displayed in Figure 4.1.5a. The target mean replicas is $\mu_{N_{s,t}} = 3.13$.

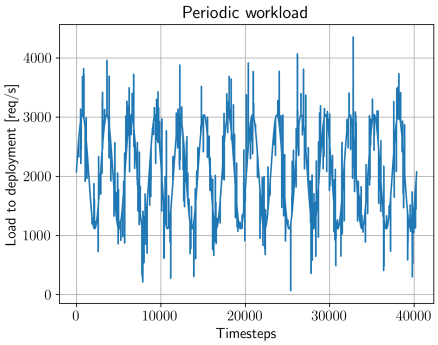
In the inference phase, we make use of a variation of the workload $\hat{\mathbb{H}}_r$, which will be referenced as $\hat{\mathbb{H}}'_r$, with a change in the spike duration, sampled from the uniform distribution $\rho'' = [10, 30]$, and the spike probability, increased to $p' = 0.02$. For the signal $\hat{\mathbb{H}}'_r$, reported in Figure 4.1.4b over one period and in Figure 4.1.5b over two weeks, the mean optimal replicas is $\mu_{N_{s,t}} = 3.08$.



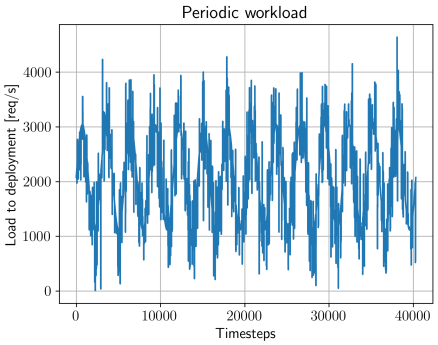
(a) One period of periodic workload with random spikes $\hat{\mathbb{H}}_r$with optimal replicas.

(b) One period of periodic workload with random spikes $\hat{\mathbb{H}}'_r$ with optimal replicas.

Figure 4.1.4: One day of periodic signals with random traffic spikes.



(a) Periodic workload with random spikes $\hat{\mathbb{H}}_r$ over two simulated weeks.



(b) Periodic workload with random spikes $\hat{\mathbb{H}}'_r$ over two simulated weeks.

Figure 4.1.5: Periodic signals with random traffic spikes over two weeks.

4.2 Comparison with the baseline autoscaler

In this last experiment, we aim at answering our research questions **RQ2** and **RQ3** by comparing the trained models with the default Kubernetes HPA on the same loads in the real cluster.

Due to time and resource constraints, we limit the experiment to 24 hours only. Nonetheless, we believe that this inference time is sufficient to gather enough data to answer our questions.

4.2.1 Procedure

In the following, we describe the procedure applied to carry out this second experiment on the cluster emulation.

4.2.1.1 Pattern generation

Differently from the simulation in the first experiment, in this scenario we do not have direct access to the load $\hat{\mathbb{H}}(t)$ expressed in req/s, rather, we can only control the number of simulated users from *loadgenerator*. Therefore, we apply the extra step to map number of users $\mathbb{U}(t)$ to the load as in Equation 4.1. The ratio $50/70$ is taken from Table 3.6.1 as to

50 users correspond about $70 req/s$.

$$\mathbb{U}(t) = \left\lceil \frac{50}{70} \hat{\mathbb{H}}(t) \right\rceil \tag{4.1}$$

Finally, in this last experiment, we do not consider the cases of a constant workload $\hat{\mathbb{H}}_c(t)$ and stepwise workload $\hat{\mathbb{H}}_s(t)$, as they are of little interest in a realistic scenario. Therefore, we proceed by testing the remaining workload patterns $\hat{\mathbb{H}}_p(t)$, $\hat{\mathbb{H}}_{pc}(t)$, $\hat{\mathbb{H}}_r(t)$, and $\hat{\mathbb{H}}'_r(t)$.

4.2.1.2 Comparison

To compare the default HPA with the agent in terms of SLA compliance and resource utilisation, we take the best performing models in terms of total accumulated reward G_T, and deploy them in the cloud environment.

The Kubernetes HPA is set only for the deployment *productcatalog* with the settings reported in Table 4.2.1. Specifically, it considers the average CPU utilisation of the deployment, computed as $C_{s,t}$, which is set to 70% as in [50]. The stabilisation window after taking a scaling action is $30s$, to be comparable with the agent's action cycle time.

`deployment`	productcatalog
`minReplicas`	1
`maxReplicas`	15
`metric`	CPU resource utilisation
`averageUtilization`	70
`stabilizationWindowSeconds`	30

Table 4.2.1: Kubernetes HPA settings.

To compare the two autoscalers, we deploy the application *Online Boutique* in two different namespaces, we set the number of replicas of the services related to *productcatalog* to 6, as described in Section 3.7, and we deploy the trained model-free agent in one namespace, while the HPA in the other. We deploy the same workload to both namespaces using *loadgenerator*.

We compute the number of SLA violations by the ratio between the number of times $L_{s,t} > \bar{L}_s$ and the number of timesteps T

elapsed since the beginning of the experiment. Regarding the resource utilisation, we compute the mean number of replicas $\mu_{N_{s,t}}$ across the whole experiment, and the mean CPU utilisation $\mu_{C_{s,t}}$ from the two autoscalers.

Chapter 5

Results and Discussion

In this Chapter, we present and discuss the results obtained from the experiments described in Chapter 4. In particular, we show that different observations spaces result in drastically different performances in terms of G_T for all workload patterns. We can identify the best and worst performers in our cases and have an intuition on which performance metric is most suitable to approximate the optimal scaling policy. In addition, we show that the model-free RL algorithm have comparable performance with respect to the Kubernetes autoscaler, and in some cases results in higher resource utilisation.

5.1 Simulation on different workload patterns

The first experiment aims to answer our main research question:

What is the impact of including specific container's performance metrics (e.g. CPU, memory, latency, RPS) in the state space on a model-free RL agent's ability to accurately predict optimal autoscaling actions?

To assess the influence, we repeatedly tested different state space definitions on various workload patterns in a simulated environment. In the following, we report the tables and plots of our measurements for each workload pattern. Due to space concerns, some Figures are

reported in Appendix 6.1.

5.1.1 Constant load

With this first pattern, we mainly wanted to assess the ability of the agent to learn an optimal policy. In the following, we report the measurements during the learning and inference phases.

5.1.1.1 Training results

State space	μ_{G_T}	σ_{G_T}
$\boldsymbol{S}_1$	-28898.11	749324.56
$\boldsymbol{S}_2$	850589.16	157746.43
$\boldsymbol{S}_3$	861235.11	398928.24
$\boldsymbol{S}_4$	1496281.52	318970.26
$\boldsymbol{S}_5$	-5530041.19	9700216.10
$\boldsymbol{S}_6$	$\mathbf{1597000.73}$	279905.55

Table 5.1.1: Comparison between mean episode rewards in the learning phase for $\hat{\mathbb{H}}_c(t)$. Means are computed across 10 trials.

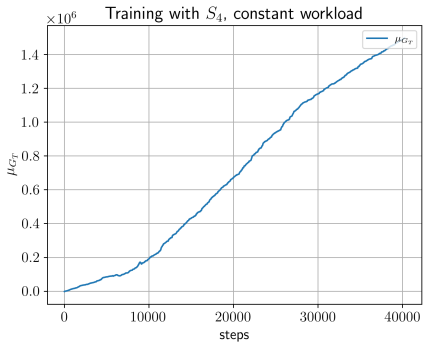
In Table 5.1.1, are reported the mean values taken at the end of the learning phase for each state $\boldsymbol{S}$. From these results, it is possible to evince that the model is in fact able to learn in some scenarios, considering the positive values of μ_{G_T}. For example, in Figure 5.1.1d, one can see the increasing trend that the accumulated reward assumes with state $\boldsymbol{S}_6$. With states $\boldsymbol{S}_1$ and $\boldsymbol{S}_5$, the reward is negative, see Figures 5.1.1a and 5.1.1c, respectively. However, while with $\boldsymbol{S}_1$ the agent shows subsets of good performance during training, with $\boldsymbol{S}_5$ the actions taken are consistently leading to negative rewards. Lastly, $\boldsymbol{S}_6$ will be the best performer in most of our experiments considering μ_{G_T}.

Most importantly, from this initial test, we can already register a difference in the agent's performance when observing different states.

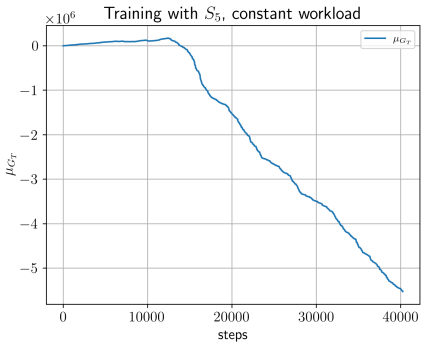
5.1.1.2 Inference results

During this phase, we select the best performing model for each state definition considering the highest G_T at the end of the episode, among

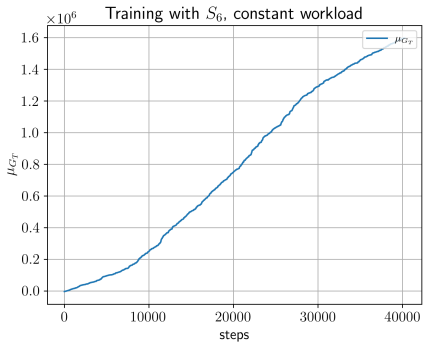
(a) Total reward G_T for state $\boldsymbol{S}_1$ on constant load.



(b) Total reward G_T for state $\boldsymbol{S}_4$ on constant load.



(c) Total reward G_T for state $\boldsymbol{S}_5$ on constant load.



(d) Total reward G_T for state $\boldsymbol{S}_6$ on constant load.

Figure 5.1.1: Training results from constant load.

State space	G_T	$\mu_{N_{s,t}}$	$\sigma_{N_{s,t}}$	SLA violations
$\boldsymbol{S}_1$	1223270.26	8.5	0.51	0.0%
$\boldsymbol{S}_2$	964104.69	7.57	0.5	0.0%
$\boldsymbol{S}_3$	1370110.07	9.0	0.07	0.0%
$\boldsymbol{S}_4$	2513303.99	3.0	0.03	0.0%
$\boldsymbol{S}_5$	1224723.14	8.5	0.5	0.0%
$\boldsymbol{S}_6$	**2513362.06**	3.0	0.03	0.0%

Table 5.1.2: Comparison of the highest episode rewards in the inference phase for $\hat{\mathbb{H}}_c(t)$.

all 10 trials. In Table 5.1.2, we see that the best performer in each state is in fact able to scale in order to avoid SLA violations.

However, not all observations lead to the optimal number of mean replicas $\mu_{N_{s,t}} = 3$. In fact, some states such as $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, $\boldsymbol{S}_3$, and $\boldsymbol{S}_5$, lead the agent to overprovision the resources, which is undesirable, and it should be limited by our designed reward function. This may be due to the fact that the agent remains stuck on a local optima, as we can see from Table 3.7.2, where relatively high rewards R_t are assigned also for $N_{s,t} = 8$ and $N_{s,t} = 9$. However, states $\boldsymbol{S}_4$ and $\boldsymbol{S}_6$, both reach the target number of replicas.

Similarly to what was registered during the training phase, we see that $\boldsymbol{S}_6$ provides the most meaningful information to the model, resulting in the highest reward, though only slightly higher than $\boldsymbol{S}_4$. From this initial assessment, we can hypothesise that $L_{s,t}$ is a key metric for policy learning with *productcatalog*.

5.1.2 Stepwise load

In this scenario, we consider a load that suddenly increases at regular intervals to verify that the model adapts its policy to a varying pattern.

5.1.2.1 Training results

State space	μ_{G_T}	σ_{G_T}
$\boldsymbol{S}_1$	-3590011.41	3552471.98
$\boldsymbol{S}_2$	851625.57	443844.67
$\boldsymbol{S}_3$	190065.55	970421.70
$\boldsymbol{S}_4$	1683659.76	1470629.83
$\boldsymbol{S}_5$	-8184169.47	10616383.40
$\boldsymbol{S}_6$	$\mathbf{2191740.12}$	806221.42

Table 5.1.3: Comparison between mean episode rewards in the learning phase for $\hat{\mathbb{H}}_s(t)$. Means are computed across 10 trials.

In Table 5.1.3, it is possible to see that once again $\boldsymbol{S}_6$ outperforms the other states. In particular, states $\boldsymbol{S}_1$ and $\boldsymbol{S}_5$ report a negative μ_{G_T}, indicating a general inability of the model to learn an optimal policy. By looking at the metrics included in these states, we infer that $H_{s,t}$,

which is the average RPS arriving to each pod, may be the cause of this unwanted behaviour, which leads to the negative reward visible in Figure 6.1.8. Please refer to the Figures in the Appendix 6.1 for the other state spaces.

5.1.2.2 Inference results

State space	G_T	$\mu_{N_{s,t}}$	$\sigma_{N_{s,t}}$	SLA violations
$\boldsymbol{S}_1$	932659.33	13.9	2.91	0.0%
$\boldsymbol{S}_2$	1625007.59	4.86	1.57	0.0%
$\boldsymbol{S}_3$	−5499265.72	3.4	1.74	39.0%
$\boldsymbol{S}_4$	**2541388.2**	3.68	0.47	0.0%
$\boldsymbol{S}_5$	−564360.44	4.88	2.83	19.0%
$\boldsymbol{S}_6$	2447887.04	3.68	0.47	0.0%

Table 5.1.4: Comparison of the highest episode rewards in the inference phase for $\hat{\mathbb{H}}_s(t)$.

In the inference phase, we selected the best performers for each state and simulated the load $\hat{\mathbb{H}}_s(t)$ on the trained models. From Table 5.1.4, it is possible to see that, once again, states $\boldsymbol{S}_4$ and $\boldsymbol{S}_6$ outperform the rest. However, in this scenario $\boldsymbol{S}_4$ provides the highest performance and is closer to the target $\mu_{N_{s,t}}$ of 3.12 replicas.

Furthermore, with this second workload, we could validate the fact that the agent is able to adapt to a dynamic workload, and we can corroborate our hypothesis on the meaningfulness of $L_{s,t}$ to discover the optimal policy, compared to other observation spaces.

5.1.3 Periodic load

This third workload signal is intended to represent more closely a realistic scenario in which users visit the website mostly during the late morning and afternoon, resulting in peak loads, than during night, resulting in the lowest traffic.

5.1.3.1 Training results

Compared to the first two signals, from the training results in Table 5.1.5 it is visible how a more dynamic workload makes the policy

State space	μ_{G_T}	σ_{G_T}
$\boldsymbol{S}_1$	-4723539.02	2399623.94
$\boldsymbol{S}_2$	-144156.15	2098845.93
$\boldsymbol{S}_3$	-126584.11	1889523.96
$\boldsymbol{S}_4$	1759449.20	735228.07
$\boldsymbol{S}_5$	-3782956.83	1223759.89
$\boldsymbol{S}_6$	$\mathbf{2080581.76}$	569356.98

Table 5.1.5: Comparison of the mean episode rewards in the learning phase for $\hat{\mathbb{H}}_p(t)$. Means are computed across 10 trials.

approximation harder for the model-free agent. In fact, four out of six state space definitions provide a negative total reward at the end of the training. Nonetheless, we note that $\boldsymbol{S}_4$ and $\boldsymbol{S}_6$ still provide positive results.

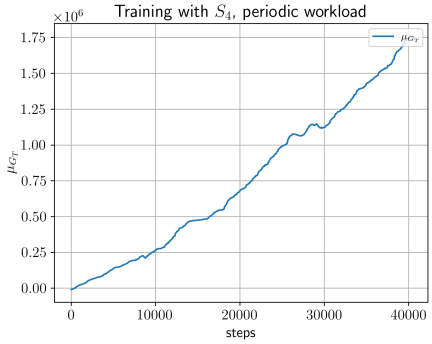
Given that $\boldsymbol{S}_4 = \{L_{s,t}, M_{s,t}, N_{s,t}\}$ and $\boldsymbol{S}_6 = \{L_{s,t}, N_{s,t}\}$, we note that the presence of $M_{s,t}$ may not influence the performance of the model considerably, while $L_{s,t}$ remains the dominant metric. In fact, by looking at Table 3.6.1, we deduce that the target backend *productcatalog* does not perform memory-intensive operations, and thus $M_{s,t}$, being on a constant level throughout the experiment, is disregarded by the model.

If we look at Figures 5.1.2a and 5.1.2b, for the total reward with $\boldsymbol{S}_4$ and $\boldsymbol{S}_5$, respectively, we register once again a poor performance due to the presence of the metric $H_{s,t}$ in $\boldsymbol{S}_5$.

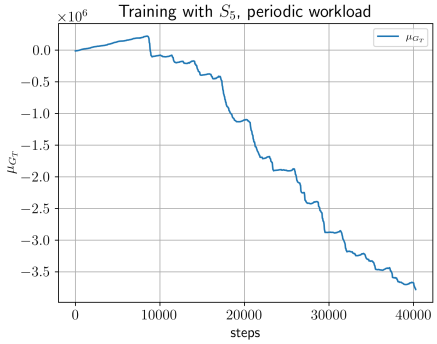
5.1.3.2 Inference results

State space	G_T	$\mu_{N_{s,t}}$	$\sigma_{N_{s,t}}$	SLA violations
$\boldsymbol{S}_1$	-71983007.99	1.0	0.02	79.0%
$\boldsymbol{S}_2$	2148645.88	3.22	1.31	11.0%
$\boldsymbol{S}_3$	982636.71	8.08	2.85	6.0%
$\boldsymbol{S}_4$	$\mathbf{2530195.22}$	3.27	0.45	20.0%
$\boldsymbol{S}_5$	-10943870.25	5.76	3.04	23.0%
$\boldsymbol{S}_6$	1266886.5	4.18	0.84	17.0%

Table 5.1.6: Comparison of the highest episode rewards in the inference phase for $\hat{\mathbb{H}}_p(t)$.



(a) Total reward G_T for state $\boldsymbol{S}_4$ on periodic load.



(b) Total reward G_T for state $\boldsymbol{S}_5$ on periodic load.

Figure 5.1.2: Training results from periodic workload.

State space	G_T	$\mu_{N_{s,t}}$	$\sigma_{N_{s,t}}$	SLA violations
$\boldsymbol{S}_1$	−71975848.72	1.0	0.02	79.0%
$\boldsymbol{S}_2$	2444890.64	3.18	1.35	9.0%
$\boldsymbol{S}_3$	1580248.55	5.63	1.65	8.0%
$\boldsymbol{S}_4$	**2530689.17**	3.27	0.44	20.0%
$\boldsymbol{S}_5$	−21917220.47	2.51	1.29	52.0%
$\boldsymbol{S}_6$	1419242.85	3.87	0.84	20.0%

Table 5.1.7: Comparison of the highest episode rewards in the inference phase for $\hat{\mathbb{H}}_{pc}(t)$.

In the inference phase, we consider the workloads $\hat{\mathbb{H}}_p(t)$ and $\hat{\mathbb{H}}_{pc}(t)$, as discussed in Chapter 4.

The results from the best performing models in the two cases are reported in Tables 5.1.6 and 5.1.7. In both scenarios, every inference from the model-free agent produces a violation of the target latency. However, we note that even though $\boldsymbol{S}_3$ provides the best performance in terms of SLA violations, it is due to the overprovisioning of resources, with the mean number of replicas as high as 8 for $\hat{\mathbb{H}}_p(t)$, around 3 times higher than the optimal $\mu_{N_{s,t}} = 2.84$. This may be due to the model being stuck on a local optima, as reported above.

On average, in our simulation, $\boldsymbol{S}_4$ obtains the highest reward at the end of the inference, even though with a higher $\mu_{N_{s,t}}$ than the target and at the price of 20% SLA violations. Lastly, we can see from our results that the agent trained on the pattern $\hat{\mathbb{H}}_p(t)$ can be deployed to infer with pattern $\hat{\mathbb{H}}_{pc}(t)$ and provide comparable results, showing an ability to generalise the learned policy to similar contexts.

5.1.4 Periodic load with random spikes

The last workload pattern includes a random change $\delta(t)$ in the number of RPS from $\hat{\mathbb{H}}_p(t)$, uniformly distributed in the interval $\rho = [-962.5, +962.5]req/s$, for a time duration randomly sampled from the interval $\rho' = [150, 600]s$. While the signal $\hat{\mathbb{H}}_r(t)$ is used for training, both $\hat{\mathbb{H}}_r(t)$ and $\hat{\mathbb{H}}'_r(t)$ are used for the inference.

5.1.4.1 Training results

State space	μ_{G_T}	σ_{G_T}
$\boldsymbol{S}_1$	-1889657.54	1490874.97
$\boldsymbol{S}_2$	-624875.54	2388503.40
$\boldsymbol{S}_3$	-1352381.22	2874619.61
$\boldsymbol{S}_4$	1841781.9	374652.58
$\boldsymbol{S}_5$	-7035480.93	4230584.29
$\boldsymbol{S}_6$	$\mathbf{1970187.88}$	193810.93

Table 5.1.8: Comparison episode rewards in the learning phase for $\hat{\mathbb{H}}_r(t)$. Means are computed across the 10 trainings.

The training results from Table 5.1.8 are similar to those reported for the periodic signal $\hat{\mathbb{H}}_p(t)$. This is expected because of the definition of $\hat{\mathbb{H}}_r(t)$, which resembles the same pattern of the previous scenario. We note that $\boldsymbol{S}_4$ and $\boldsymbol{S}_6$ both provide positive outcomes. Interestingly, in Figure 5.1.3a, the trend for μ_{G_t} shows an initial increase, followed by an abrupt decrease after the first third of training, followed by an increasing trend. This behaviour indicates that in the exploration phase, on average across 10 trials, the agent remained stuck in non-valuable state S_t, to later approximate a better policy. The introduction of a random action selection every 1000 steps in our Algorithm 2, may be the reason of its movement from local minima towards more valuable states. Even though $\boldsymbol{S}_2$ provides a negative mean total reward at the end of the episode, its trend toward the end of the training is comparable with the good approximation obtained with $\boldsymbol{S}_4$, in Figure 5.1.3b.

(a) Total reward G_T for state $\boldsymbol{S}_2$ on periodic with random spikes load.

(b) Total reward G_T for state $\boldsymbol{S}_4$ on periodic with random spikes load.

Figure 5.1.3: Training results for periodic workload with random spikes.

5.1.4.2 Inference results

As visible in Table 5.1.9, we register a positive accumulated reward from the best trained model in every state space definition. Interestingly, although the highest rewards G_T are obtained by $\boldsymbol{S}_4$ and $\boldsymbol{S}_6$, these are also the states with highest SLA violation, 24% and 39%, respectively. The reason may lay in the fact that sudden increases in the load due to traffic spikes may require more than one scaling cycle to reach the optimal $N_{s,t}$. This issue is inherited from the reactive nature of

State space	G_T	$\mu_{N_{s,t}}$	$\sigma_{N_{s,t}}$	SLA violations
$\boldsymbol{S}_1$	1011300.96	10.49	2.12	0.0%
$\boldsymbol{S}_2$	1361218.23	4.81	1.09	0.0%
$\boldsymbol{S}_3$	505651.92	7.76	1.4	0.0%
$\boldsymbol{S}_4$	**2199671.47**	3.16	0.36	28.99%
$\boldsymbol{S}_5$	823892.81	15.0	0.15	0.0%
$\boldsymbol{S}_6$	1735559.63	3.01	0.2	42.0%

Table 5.1.9: Comparison of the highest episode rewards in the inference phase for $\hat{\mathbb{H}}_r(t)$.

State space	G_T	$\mu_{N_{s,t}}$	$\sigma_{N_{s,t}}$	SLA violations
$\boldsymbol{S}_1$	990837.12	10.43	2.3	0.0%
$\boldsymbol{S}_2$	1315011.52	5.02	1.35	0.0%
$\boldsymbol{S}_3$	579443.07	7.71	1.54	0.0%
$\boldsymbol{S}_4$	**2401394.24**	3.19	0.39	24.0%
$\boldsymbol{S}_5$	838240.89	14.99	0.19	0.0%
$\boldsymbol{S}_6$	1825204.69	3.03	0.21	39.0%

Table 5.1.10: Comparison of the highest episode rewards in the inference phase for $\hat{\mathbb{H}}'_r(t)$.

the autoscaling approach and has been discussed thoroughly in the literature [50], [68], [70]. Moreover, as discussed in [70], having an agent capable of changing the number of replicas by one unit only, may be a limiting factor for its performance.

In addition, $\boldsymbol{S}_2$ can be considered a reasonable model in both cases even though it does not register the highest G_T. In fact, it is the state space definition that in the simulation for the inference provides the closest $\mu_{N_{s,t}}$ to the optimal one with 0.0% SLA violations. Other models, such as $\boldsymbol{S}_1$ and $\boldsymbol{S}_5$ provide the same performance in terms of latency violations but at the price of a consistent overprovisioning of resources.

During this inference phase, we could assess that the agent trained on $\hat{\mathbb{H}}_r(t)$ could be used to infer on a much more dynamic workload $\hat{\mathbb{H}}'_r(t)$. Since the results from the two scenarios are comparable, we can conclude that our model-free agent is capable of generalising its learned policy to other scenarios.

Finally, also in this last test we register differences in the agent's performance due to different state space definitions. In particular, from our empirical results, we could show that the metrics $N_{s,t}$, representing the number of pods, and $L_{s,t}$, to represent the 95^{th} percentile latency, are the most important for policy approximation. In fact, these metrics are the only one specifically used in the designed reward function $R(N_{s,t}, L_{s,t})$, indicating that the model-free agent provides the best performance with only those metrics defined to reach its optimisation goal. With these results, we can indeed conclude on our **RQ1**, showing that there is an influence on the state space definitions on the autoscaling policy and, in particular, additional performance metrics in the observation, other than those present in the reward function, can be considered superfluous.

5.2 Comparison with the baseline autoscaler

This second and last experiment is aimed at answering our remaining research questions:

How does the use of average CPU utilisation impact the QoS and the resource utilisation in the baseline

autoscaler?

How does the model-free algorithm applied to the autoscaler perform compared to the baseline in terms of SLA assurance and resource utilisation?

Both autoscalers have been deployed in the same GKE cluster for 24 hours, on different namespaces. In Table 5.2.1, we reported the results obtained at the end of the inference period from the comparison between our best performing agents and the Kubernetes HPA set on 70% average CPU utilisation.

Load pattern	Autoscaler	SLA violations	$\mu_{N_{s,t}}$	$\mu_{C_{s,t}}$
Periodic load $\hat{\mathbb{H}}_p$	Agent on $\boldsymbol{S}_4$	18.03%	3.19	68.93%
	HPA	$\mathbf{0.0\%}$	3.92	60.32%
Periodic load $\hat{\mathbb{H}}_{pc}$	Agent on $\boldsymbol{S}_4$	34.67%	3.36	64.41%
	HPA	$\mathbf{0.0\%}$	3.77	58.11%
Periodic with spikes $\hat{\mathbb{H}}_r$	Agent on $\boldsymbol{S}_2$	0.55%	3.78	71.60%
	Agent on $\boldsymbol{S}_4$	3.61%	3.23	80.67%
	HPA	$\mathbf{0.0\%}$	4.54	65.03%
Periodic with spikes $\hat{\mathbb{H}}'_r$	Agent on $\boldsymbol{S}_2$	$\mathbf{0.69\%}$	4.09	69.59%
	Agent on $\boldsymbol{S}_4$	8.87%	3.30	78.75%
	HPA	0.83%	4.54	63.36%

Table 5.2.1: Comparison of the best performing agents with the Kubernetes HPA on 24 hours of inference.

Because of the lengthy inference time, and the cost of resources deployed in the public cloud, for our experiment we only selected those workload patterns that make more sense in a practical scenario, namely, $\hat{\mathbb{H}}_p(t)$, $\hat{\mathbb{H}}_{pc}(t)$, $\hat{\mathbb{H}}_r(t)$, and $\hat{\mathbb{H}}'_r(t)$. Then, for each workload we selected the best performing models based on the total accumulated reward G_T obtained in the simulation, and deployed those on the actual cluster.

5.2.0.1 Inference on periodic loads

With respect to the periodic pattern $\hat{\mathbb{H}}_p(t)$ with the agent trained on $\boldsymbol{S}_4$, by comparing the SLA violations, the model-free agent violates the latency agreement 18.03% of the times, showing a worse performance compared to the Kubernetes HPA, which does not violate

the agreement. However, we can see that the average CPU utilisation $C_{s,t}$ of the HPA is lower than the target of 70%. This indicates that the Kubernetes autoscaler is not always able to provide the desired resource utilisation. Lastly, we note that both autoscalers deploy a mean number of replicas higher than the optimal computed in our simulation $\mu_{N_{s,t}} = 2.84$.

Concerning the load with cosine shape $\hat{\mathbb{H}}_{pc}(t)$ on the agent with observation space $\boldsymbol{S}_4$, we record the worse performance in terms of SLA violations, set at 34.67%, while the baseline is set to 0.0%. In terms of resource utilisation, we notice that the HPA uses on average the lowest amount of resource, with $\mu_{C_{s,t}} = 58.11\%$, which is 11.89% lower than the target of 70%.

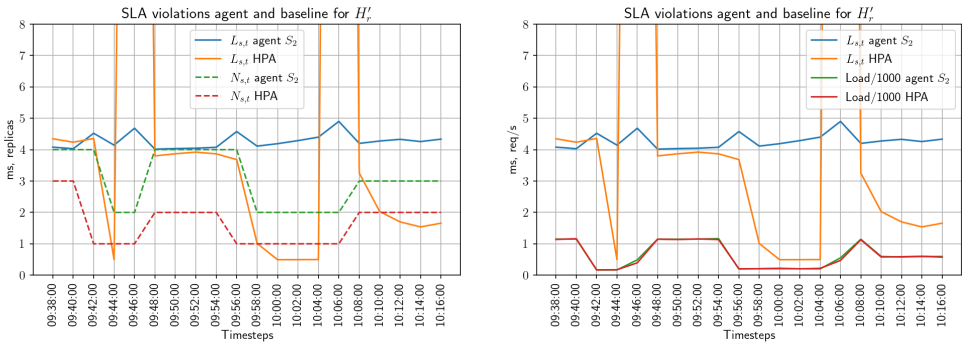
5.2.0.2 Inference on periodic loads with random spikes

Considering the periodic patterns with random traffic spikes $\hat{\mathbb{H}}_r(t)$, and $\hat{\mathbb{H}}'_r(t)$ we decided to test the agents on two state space definitions, with the goal of obtaining more data to compare the two autoscalers.

Starting with the agent observing $\boldsymbol{S}_2$, which includes all performance metrics except the mean RPS to the pod, we register the best performance in terms of SLA violations among all agents deployed in GKE, which is 0.55%. Moreover, its performance can be considered reasonably good if we compare the mean CPU utilisation, which is higher with respect to the HPA by 6.57% on average. Most notably, we can see that the agent on the observation space $\boldsymbol{S}_2$ deployed with $\hat{\mathbb{H}}'_r(t)$ performs better than the Kubernetes HPA. In terms of SLA violations, the performance of the model-free agent is 0.69%, while the HPA registers 0.83% violations. Moreover, considering the resource utilisation, the agent sets $\mu_{C_{s,t}} = 69.59\%$, while the baseline $\mu_{C_{s,t}} = 63.36\%$, indicating that in this case the agent is better at exploiting the available CPU to meet the SLA. Compared to the optimal number of replicas $\mu_{N_{s,t}} = 3.13$ for $\hat{\mathbb{H}}_r(t)$ and $\mu_{N_{s,t}} = 3.08$ for $\hat{\mathbb{H}}'_r(t)$, we see that both agent and baseline on average use a higher number of replicas throughout the experiment.

In fact, this is the only case in which we recorded an SLA violation from the Kubernetes HPA in our measurements. From Figure 5.2.1, we can see a sample of about $40'$ from the signals during the inference phase. Specifically, we reported two cases in which the Kubernetes

HPA violated the SLA while the agent did not. In Figure 5.2.1a, the HPA violates the agreement on $\bar{L}_s = 5ms$ when it sets the number of replicas to 1, while the agent keeps a minimum of 2 replicas in those scenarios of low requests, see Figure 5.2.1b.



(a) Latency $L_{s,t}$ and replicas $N_{s,t}$

(b) Latency $L_{s,t}$ and load $\hat{H}_{s,t}$

Figure 5.2.1: Comparison on SLA violations between the trained agent and the baseline.

When the agent observes state $\boldsymbol{S}_4$, we register a worse performance compared with the previous state in both $\hat{\mathbb{H}}_r(t)$ and $\hat{\mathbb{H}}'_r(t)$, which is expected since we selected this state based on G_T even though its SLA violation was high in our simulation, see Tables 5.1.9 and 5.1.10. In fact, the deployment of these agents results in 3.61% violations with $\hat{\mathbb{H}}_r(t)$ and 8.87% for $\hat{\mathbb{H}}'_r(t)$. Even though the mean CPU utilisation is highest with these observations, the performance is to be considered very negative. Lastly, note that the trade-off between SLA violations and resource utilisation could have been reduced by selecting a less restrictive parameter α in the reward function of Equation 3.2, thus leading to agent decisions that would perhaps decrease the SLA violations at the price of lower CPU utilisation.

Finally, compared to the baseline, both agents on the observation space $\boldsymbol{S}_4$ register a worse performance. And, once again, in terms of resource utilisation we see that the Kubernetes HPA leads to an average utilisation 4.97% and 6.64% lower than the target for $\hat{\mathbb{H}}_r(t)$ and $\hat{\mathbb{H}}'_r(t)$, respectively.

Load pattern and agent	Environment	SLA violations	$\mu_{N_{s,t}}$
Periodic load $\hat{\mathbb{H}}_p$, agent on $\boldsymbol{S}_4$	GKE	18.03%	3.19
	Local	22%	3.23
Periodic load $\hat{\mathbb{H}}_{pc}$, agent on $\boldsymbol{S}_4$	GKE	34.67%	3.36
	Local	22%	3.23
Periodic with spikes $\hat{\mathbb{H}}_r$, agent on $\boldsymbol{S}_2$	GKE	0.55%	3.78
	Local	0.0%	4.6
Periodic with spikes $\hat{\mathbb{H}}_r$, agent on $\boldsymbol{S}_4$	GKE	3.61%	3.23
	Local	27.0%	3.11
Periodic with spikes $\hat{\mathbb{H}}'_r$, agent on $\boldsymbol{S}_2$	GKE	0.69%	4.09
	Local	0.0%	4.52
Periodic with spikes $\hat{\mathbb{H}}'_r$, agent on $\boldsymbol{S}_4$	GKE	8.87%	3.30
	Local	27.0%	3.1

Table 5.2.2: Comparison of agent performances between simulation and emulation environments on 24 hours of inference.

5.2.1 Validation in emulated environment

Another important result taken from this experiment is an assessment on the validity of our simulated environment. If the results obtained in the emulated GKE environment and in the local simulation are generally similar, we can conclude that the applied methodology and the modelling of our system are correct. In fact, considering the performance of our best models in the simulation, then deployed in the emulated environment in the cloud, we recognise that their performances are comparable in most of the cases, and we will discuss these results in the following. Note that to make the inferences between the simulation and the emulation comparable over 24 hours, we run again the simulation on $T = 2880$ timesteps and registered $\mu_{N_{s,t}}$ and SLA violations, reported in Table 5.2.2.

If we look at the agent with observation space $\boldsymbol{S}_4$ on $\hat{\mathbb{H}}_p$, the values of $\mu_{N_{s,t}}$ are 3.23 and 3.19, for the simulation and emulation, respectively, which indicates a similar behaviour from the same agent on the two environments. In terms of SLA violations, the difference between GKE and the local environment is close to 4%.

Comparing the agent on $\hat{\mathbb{H}}_{pc}$ with observation $\boldsymbol{S}_4$, we register the same values in the local environment as in the previous case. This is expected since the difference between the signals $\hat{\mathbb{H}}_p$ and $\hat{\mathbb{H}}_{pc}$ is simply

a shift in the periodicity. However, if we look at the value of SLA violations, in the emulation it is higher than the local environment. As visible in Figure 5.2.2, we see that this is due to the fact that the load produced by *loadgenerator* to the application was too high for the range $[3500, 4000] req/s$, and the requests in the peak were most probably blocked by some bottlenecks such as *frontend*, resulting in higher latency. Unfortunately, this situation prevented us to gather more reliable data on the performance of our target service *productcatalog* for this scenario.

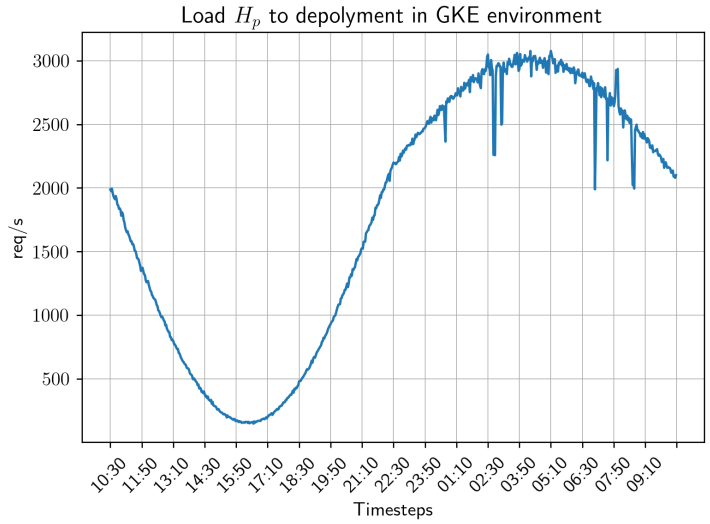
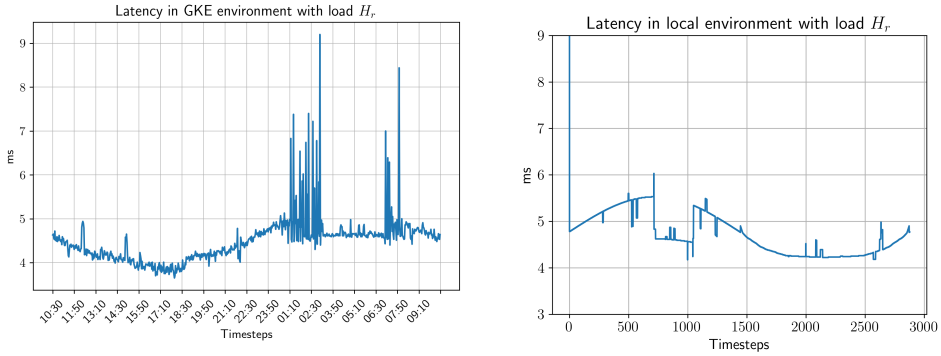


Figure 5.2.2: Load $\hat{\mathbb{H}}_p$ in GKE environment.

When tested on $\hat{\mathbb{H}}_r$, the agent with observation $\boldsymbol{S}_2$ deploys on average 4.6 replicas in the simulation and 3.78 in the emulation, while with $\boldsymbol{S}_4$ the agent exploits on average 3.11 in the simulation, and 3.23 in the emulation. However, if we look at the SLA violations, in the case of $\hat{\mathbb{H}}_r$ the results are not comparable for the agent trained on $\boldsymbol{S}_4$. In fact, contrary to the SLA violations for $\boldsymbol{S}_2$, which are 0.0% in the simulation and 0.55% in the emulation, on $\boldsymbol{S}_4$ these are 27% and 3.61% in the simulation and emulation, respectively. The reason for the unexpected good performance in the emulation is that the latency has an oscillatory behaviour around $\bar{L}_s$, resulting in more samples in which the backend is satisfying the SLA on the emulation. This is visible in Figure 5.2.3, where we compare the latency signals recorded from the emulation and from the simulation. While in the simulation the

latency is hard-coded to a fixed value which remains above $\bar{L}_s$ for many samples, around timestep $T = 500$ in Figure 5.2.3b, in the emulation the latency actually has a more oscillatory behaviour, as visible from the sample in Figure 5.2.4. Therefore, we attribute the higher percentage of SLA violations in the simulation to the fact that our modelling of the cloud environment could not replicate in details the actual latency oscillations when *productcatalog* is stressed. Nonetheless, with the agent on observation $\boldsymbol{S}_4$, we register a similar performance in terms of $\mu_{N_{s,t}}$.



(a) Latency from emulated environment.

(b) Latency from simulated environment.

Figure 5.2.3: Comparison on latencies between simulation and emulation with load $\hat{\mathbb{H}}_r$ and agent on $\boldsymbol{S}_4$.

For what concerns the agents on load $\hat{\mathbb{H}}'_r$, we record similar performances in terms of mean replicas and SLA violations, except from the agent with observation $\boldsymbol{S}_4$, where in the local environment we register higher latency for the same reasons discussed above.

In addition, the deviation in the results from the comparison between the simulation and emulation may be also be affected by the sampling difference between the data downloaded from Grafana, with 2 minutes granularity, and the one obtained from the simulation, with $30s$ granularity.

Lastly, with the results from this Section, we can indeed conclude on our research questions **RQ2** and **RQ3**. In particular, from Table 5.2.1, for **RQ2** we can assess the impact of the use of average CPU as triggering metric on the QoS and resource utilisation in the baseline autoscaler. Already from the results obtained from our first

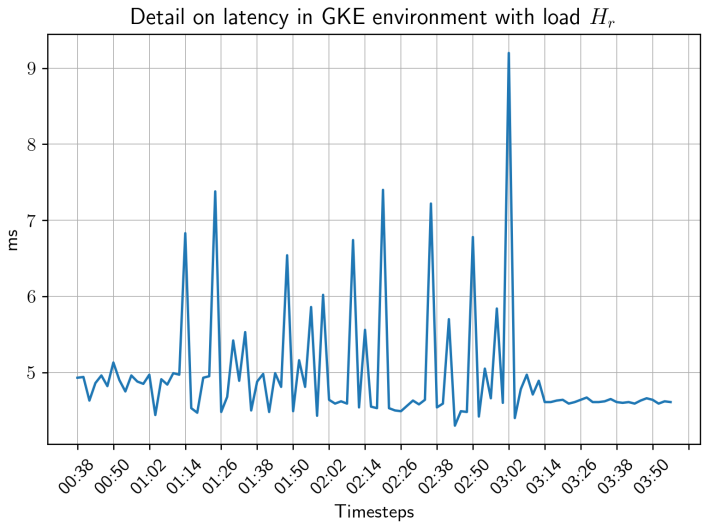


Figure 5.2.4: Detail on latency fluctuation in GKE environment.

measurements in Table 3.6.1, we could expect that the choice of average CPU utilisation set to 70% would have led to a good performance of the autoscaler, since the service *productcatalog* showed a performance degradation at around $C_{s,t} = 80\%$. In fact, the baseline autoscaler provided 0.0% SLA violations in most cases, except in highly dynamic loads. However, in each measurement of the Kubernetes HPA, we registered a lower resource utilisation than the target, from about 5% to 11% less than the threshold. This highlights a limitation of the baseline, which is not able to always provide the optimal resource utilisation because of the simplicity of its control loop algorithm. In addition, as in the case of the model-free agent, the baseline always uses a higher number of replicas than the optimal we calculated. Therefore, we conclude that in the case of a CPU-intensive microservice, the use of average CPU utilisation as triggering metric in the HPA provides reasonably good performance in terms of SLA violations, even though the information on the latency is not explicitly provided to the autoscaler. However, this performance is highly correlated to the fact that our target backend service showed a behaviour that best suited the baseline in our measurements. In fact, as it has been shown in the literature, the choice of that infrastructure-level metric may not be suitable in other cases, such as memory-intensive applications. We leave the assessment of memory-intesive backend services in *Online*

Boutique to future work.

For what concerns **RQ3**, our assessment of model-free DRL algorithms compared to the Kubernetes autoscaler can be derived again from the results reported in Table 5.2.1. Specifically, we recorded a generally worse performance of the DRL agent with respect to the HPA in terms of SLA violations, with higher resource utilisation than the baseline. However, considering that the main goal of the agent was to satisfy the SLA of a target p_{95} below $5ms$ for the whole control task, with the periodic load the baseline was considerably better than the agent. However, when we considered our more realistic workload pattern with random traffic spikes, in one case we recorded a better performance of the model-free agent compared to the baseline in terms of both SLA violations and resource utilisation. Even though the main focus of our research was to assess the influence of various observation spaces on the policy optimisation capabilities of model-free agents, and despite the simplifications and approximations that we engineered in our systems, we could show with our results that model-free Actor-Critic DRL models can in fact provide a performance comparable to the one of the baseline. Moreover, the potential of our algorithm lies in the fact that (1) it was able to autonomously learn how to scale a target microservice without prior knowledge about the system, (2) that it can be adapted to various types of deployments, and (3) that it can be used with different workload patterns.

Chapter 6

Conclusions

Cloud computing has significantly contributed to the growth and success of IT enterprises, offering efficiency and rapid innovation in today's fast-paced economic and technological landscape. It relies on virtualisation technologies such as software containers to meet the high flexibility and scalability demands of modern applications. Consequently, to efficiently manage containerised applications at scale, container orchestrators such as Kubernetes have become essential. Nevertheless, optimising the performance of these orchestrators is challenging due to the dynamic nature of containerised environments and the diverse requirements of modern implementations. The use of Actor-Critic algorithms for DRL has proved to be a viable solution in container orchestration, with multiple successful examples of their application in the recent research.

Despite the existence of solutions that combine various metrics to evaluate autoscaling actions, there has been a lack of examination regarding how model-free RL algorithms interact with a state space based on those metrics. Thus, the objective of this degree project has been to explore the effectiveness of model-free Actor-Critic methods, specifically Advantage Actor Critic (A2C), concerning multiple performance metrics in the context of horizontal autoscaling in Kubernetes.

From our findings, we can conclude on our research questions in the following way. For what concerns the first, we assessed the influence of different state space definitions on the agent's ability to approximate

a good policy by considering the total accumulated reward at the end of the training period. In fact, we registered different behaviours for different sets of metrics. Specifically, those sets that only included the parameters present in the reward function, such as $N_{s,t}$ and $L_{s,t}$, proved to be the most effective. This is in fact reasonable, as the complexity of the approximation also increases with more input variables. In addition, this result is interesting if we compare it to *me-kube* [50], for example. In fact, their model-free agent was trained on multiple metrics, even though the objective given to the agent was to be SLA compliant in terms of service latency, as in our case. Their model-free agent performed poorly compared with the model-based agent, with 64% violations and 14% violations, respectively. However, in our experiments, we showed that with a dynamic workload pattern, we could achieve down to 0.55% SLA violations by selecting the right metric on a model-free agent, and in one case perform even better than the baseline.

Concerning our second research question, we assessed that in fact, the Kubernetes HPA performed well in terms of QoS by always avoiding the SLA violation, except with a highly dynamic load. However, we have to consider that it was tested on a CPU-intensive service, *productcatalog*, which in fact showed performance degradation with more than 80% average CPU utilisation, while 70% was used as trigger for the baseline. Another consideration regarding the baseline autoscaler is the fact that, despite its settings on 70% CPU utilisation, it always exploited the resources on 5% to 11% less utilisation than the target, indicating a general inability to precisely meet the goal in every scenario. Despite this, the Kubernetes HPA proves to be a very simple but viable solution for container autoscaling in MSA applications for cases in which most of the tasks are CPU-intensive.

Finally, comparing the model-free solution with the baseline autoscaler in Kubernetes, we can say that, in terms of SLA assurance, the agent performed worse, in one case with up to 18% more violations. However, if we consider that the agent was trained solely on the simulation, we believe that a more sophisticated simulation with higher sample rate, or even the possibility to train at a cheap cost the agent directly in the real environment, could easily outperform the baseline. In fact, our results show that in some cases the violations were close to 0% with fewer replicas used compared to the baseline. Moreover, in a

periodic workload with random traffic spikes, our model-free agent was able to beat the baseline in terms of both SLA assurance and resource utilisation. Lastly, the advantage of a model-free agent is also its ability to autonomously learn the autoscaling policy when given the correct observation space, and autonomously understand the best scaling thresholds, without relying on manually coded values for resources utilisation triggers such as vCPU and vMemory in the Kubernetes autoscalers.

Concerning our methodology, we believe that we applied an appropriate method considering the limitations of our hardware and resources. Moreover, we did our best to make our experiments reproducible by providing all the instructions to produce a similar implementation. We also believe that the methodology applied for the model training, which mainly involved a simulation to reduce the monetary and time costs by running it on a local environment, can be useful to other similar scenarios, ultimately providing the benefits of an autonomous autoscaler at a reasonable cost. With our last experiment, we could also show that agents' behaviours trained in the simulated environment were generally comparable to those of the same agents deployed in the emulated cloud environment, even though the stability of the system was hampered by the peak load in two scenarios. Therefore, we can conclude that our methodology proved to be effective in enabling us to gather the data we needed to answer our three research questions and critically analyse the results.

Lastly, we believe that our findings will help researchers in making better-informed decisions about the behaviour of model-free Actor-Critic algorithms in dynamic environments such as MSA application deployed in the cloud, by identifying appropriate metrics to assess service performance and train model-free agents.

6.1 Future Work

There are various ways in which the depth of this thesis could have been improved, but, mainly due to time concerns, we decided to leave that to future works. In the following, we briefly observe and suggest some possible improvements.

Firstly, we considered the performance of Actor-Critic methods only

by deploying one of its possible implementations, that is A2C. In the future, we could compare the performance of different algorithms on various state space definitions as we did in this degree project, thus providing results to take even better decisions when designing an implementation involving model-free Actor-Critic methods.

Secondly, for simplicity, we considered a fully-decentralised solution in which each agent controls a single deployment in Kubernetes. In a real production environment, this may limit the scalability of the application since the number of agents would linearly increase as the number of deployments. In the future, one could implement a solution making use of a centralised solution with a single agent trained to control various components and understand their dependencies, or evaluate hybrid solutions such as hierarchical scaling.

Moreover, as we evinced from the state-of-the-art in autonomous container orchestration, the ability to predict the incoming load to the application can improve the performance of the scaling action. Therefore, perhaps the inclusion of a prediction on $\hat{H}_{s,t}$, could have further improved the performance of our model-free agent with respect to the baseline, and could have significantly changed our results since we showed that in our case $H_{s,t}$ was the metric that lead to the worst performance. In fact, we remark on the fact that we showed the importance of the selection of the most appropriate metric to scale based on the objective formulated as reward function.

Lastly, in future works, one would expect to see a fully automated machine learning pipeline that could observe data from the microservice stress tests, automate the learning process of DRL algorithms on different sets of metrics for each service, and subsequently deploy trained models directly in the production cluster. In fact, in the cloud industry, the trend of adopting ML techniques to solve various tasks in the cloud, especially container autoscaling, is gaining popularity. This is because of its clear advantages compared to traditional methods, and, most importantly, in a time when the environmental sustainability of the Internet industry becomes a major concern, being able to efficiently manage the available resources in the cloud is key to reduce its carbon emissions.

References

[1] "1.3 What are Hypervisors?" [Online]. Available: https://docs.oracle.com/cd/E50245_01/E50249/html/vmcon-hypervisor.html (visited on 05/29/2023).

[2] "2022 Service Mesh Adoption Survey," Solo.io, [Online]. Available: https://www.solo.io/resources/report/2022-service-mesh-adoption-survey/ (visited on 05/30/2023).

[3] "About Anthos Service Mesh | Google Cloud," [Online]. Available: https://cloud.google.com/service-mesh/docs/overview (visited on 07/11/2023).

[4] Alshuqayran, N., Ali, N., and Evans, R., "A Systematic Mapping Study
in Microservice Architecture," in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, Nov. 2016, pp. 44–51. doi: 10.1109/SOCA.2016.15.

[5] Balla, D., Simon, C., and Maliosz, M., "Adaptive scaling of Kubernetes pods," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, Apr. 2020, pp. 1–5. doi: 10.1109/NOMS47738.2020.9110428.

[6] Bao,
Y., Peng, Y., and Wu, C., "Deep Learning-based Job Placement in Distributed Machine Learning Clusters," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, Paris, France: IEEE, Apr. 2019, pp. 505–513, isbn: 978-1-72810-515-4. doi: 10.1109/INFOCOM.2019.8737460. [Online]. Available: https://ieeexplore.ieee.org/document/8737460/ (visited on 05/24/2023).

[7] Box (Google), C. “Introducing istiod: Simplifying the control plane,” Istio, [Online]. Available: https://istio.io/v1.16/blog/2020/istiod/ (visited on 07/11/2023).

[8] “Boyscout99/microservice-demo: Online Boutique with an RL agent,” GitHub, [Online]. Available: https://github.com/boyscout99/microservice-demo (visited on 04/26/2023).

[9] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. “OpenAI Gym.” arXiv: 1606.01540 [cs]. (Jun. 5, 2016), [Online]. Available: http://arxiv.org/abs/1606.01540 (visited on 07/17/2023), preprint.

[10] Cailliau, E., Aerts, N., Noterman, L., and Groote, L., “A comparative study on containers and related technologies,” Nov. 20, 2016.

[11] Casalicchio, E. and Perciballi, V., “Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics,” in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, Sep. 2017, pp. 207–214. doi: 10.1109/FAS-W.2017.149.

[12] “Containerd,” [Online]. Available: https://containerd.io/ (visited on 05/29/2023).

[13] “Controllers,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/architecture/controller/ (visited on 06/04/2023).

[14] “Deployments,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (visited on 06/04/2023).

[15] “Docker Index: Dramatic Growth in Docker Usage Affirms the Continued Rising Power of Developers | Docker.” (Jul. 30, 2020), [Online]. Available: https://www.docker.com/blog/docker-index-dramatic-growth-in-docker-usage-affirms-the-continued-rising-power-of-developers/ (visited on 05/30/2023).

[16] “Docker overview,” Docker Documentation. (May 27, 2023), [Online]. Available: https://docs.docker.com/get-started/overview/ (visited on 05/29/2023).

[17] Engineering, S. “Fleet Management at Spotify (Part 2): The Path to Declarative Infrastructure,” Spotify Engineering. (May 3, 2023), [Online]. Available: https://engineering.atspotify.com/2023/05/fleet-management-at-spotify-part-2-the-path-to-declarative-infrastructure/ (visited on 05/31/2023).

[18] “Envoy Proxy - Home,” [Online]. Available: https://www.envoyproxy.io/ (visited on 03/16/2023).

[19] “FAQ,” etcd, [Online]. Available: https://etcd.io/docs/v3.5/faq/ (visited on 06/04/2023).

[20] *GoogleCloudPlatform/microservices-demo*, Google Cloud Platform, May 30, 2023. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo (visited on 05/30/2023).

[21] He, Q., Moayyedi, A., Dán, G., Koudouridis, G. P., and Tengkvist, P., “A meta-learning scheme for adaptive short-term network traffic prediction,” *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 10, pp. 2271–2283, 2020. doi: 10.1109/JSAC.2020.3000408.

[22] “Horizontal Pod Autoscaling,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (visited on 06/08/2023).

[23] “Installing Addons,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/addons/ (visited on 06/05/2023).

[24] “Introduction to Grafana | Grafana documentation,” Grafana Labs, [Online]. Available: https://grafana.com/docs/grafana/latest/introduction/ (visited on 07/11/2023).

[25] Jawaddi, S. N. A., Johari, M. H., and Ismail, A., “A review of microservices autoscaling with formal verification perspective,” *Software: Practice and Experience*, vol. 52, no. 11, pp. 2476–2495, 2022, issn: 1097-024X. doi: 10.1002/spe.3135. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3135 (visited on 02/16/2023).

[26] “Kube-apiserver,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/ (visited on 06/04/2023).

[27] "Kube-proxy," Kubernetes, [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/ (visited on 06/04/2023).

[28] "Kubelet," Kubernetes, [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/ (visited on 06/04/2023).

[29] "Kubernetes Components," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/ (visited on 06/03/2023).

[30] *Kubernetes Metrics Server*, Kubernetes SIGs, Jun. 5, 2023. [Online]. Available: https://github.com/kubernetes-sigs/metrics-server (visited on 06/05/2023).

[31] Li, W., Lemieux, Y., Gao, J., Zhao, Z., and Han, Y., "Service Mesh: Challenges, State of the Art, and Future Research Opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, Apr. 2019, pp. 122–1225. doi: 10.1109/SOSE.2019.00026.

[32] "Life of a Request — envoy 1.26.2-4aa28d documentation," [Online]. Available: https://www.envoyproxy.io/docs/envoy/v1.26.2/intro/life_of_a_request (visited on 07/11/2023).

[33] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. "Asynchronous Methods for Deep Reinforcement Learning." version 2. arXiv: 1602.01783 [cs]. (Jun. 16, 2016), [Online]. Available: http://arxiv.org/abs/1602.01783 (visited on 07/04/2023), preprint.

[34] Nachum, O., Norouzi, M., Xu, K., and Schuurmans, D. "Bridging the Gap Between Value and Policy Based Reinforcement Learning." arXiv: 1702.08892 [cs, stat]. (Nov. 22, 2017), [Online]. Available: http://arxiv.org/abs/1702.08892 (visited on 07/10/2023), preprint.

[35] "Namespaces," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/ (visited on 06/04/2023).

[36] Nguyen, T.-T., Yeom, Y.-J., Kim, T., Park, D.-H., and Kim, S., "Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration," *Sensors*, vol. 20, no. 16, p. 4621, Aug. 2020. doi: 10.3390/s20164621. [Online]. Available: https://www.mdpi.com/1424-8220/20/16/4621 (visited on 01/30/2023).

[37] "OpenAI Baselines: ACKTR & A2C," [Online]. Available: https://openai.com/research/openai-baselines-acktr-a2c (visited on 05/20/2023).

[38] "Optimize Pod autoscaling based on metrics | Kubernetes Engine," Google Cloud, [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/tutorials/autoscaling-metrics (visited on 03/02/2023).

[39] *Overview*, Kubernetes, Jun. 5, 2023. [Online]. Available: https://github.com/kubernetes/kube-state-metrics (visited on 06/05/2023).

[40] "Overview," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/overview/ (visited on 05/26/2023).

[41] "Overview," [Online]. Available: https://linkerd.io/2.13/overview/ (visited on 07/11/2023).

[42] "Pod Lifecycle," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/ (visited on 06/05/2023).

[43] "Pods," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on 06/04/2023).

[44] Prometheus. "Data model | Prometheus," [Online]. Available: https://prometheus.io/docs/concepts/data_model/ (visited on 07/11/2023).

[45] Prometheus. "Jobs and instances | Prometheus," [Online]. Available: https://prometheus.io/docs/concepts/jobs_instances/ (visited on 07/11/2023).

[46] Prometheus. "Metric types | Prometheus," [Online]. Available: https://prometheus.io/docs/concepts/metric_types/ (visited on 07/11/2023).

[47] Prometheus. “Overview | Prometheus,” [Online]. Available: https://prometheus.io/docs/introduction/overview/ (visited on 07/11/2023).

[48] Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z. T., and Iyer, R. K., “FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices,” 2020.

[49] Rossi, F., “Auto-scaling Policies to Adapt the Application Deployment in Kubernetes,” presented at the Central-European Workshop on Services and Their Composition, 2020. [Online]. Available: https://www.semanticscholar.org/paper/Auto-scaling-Policies-to-Adapt-the-Application-in-Rossi/f9220fab57c32b58ff6bc8c38289ff3b1056169f (visited on 02/05/2023).

[50] Rossi, F., Cardellini, V., and Presti, F. L., “Hierarchical Scaling of Microservices in Kubernetes,” in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, Aug. 2020, pp. 28–37. doi: 10.1109/ACSOS49614.2020.00023.

[51] “Scale a StatefulSet,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/tasks/run-application/scale-stateful-set/ (visited on 07/03/2023).

[52] “Service,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 06/04/2023).

[53] Siddiqui, T., Siddiqui, S. A., and Khan, N. A., “Comprehensive Analysis of Container Technology,” in *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*, Nov. 2019, pp. 218–223. doi: 10.1109/ISCON47742.2019.9036238.

[54] “Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations — Stable Baselines3 2.1.0a0 documentation,” [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/ (visited on 07/10/2023).

[55] Storment, J. R. and Fuller, M., *Cloud FinOps*, Second edition. O’Reilly, Jan. 18, 2023, isbn: 978-1-4920-5462-7.

[56] “Sustainability,” Google Cloud, [Online]. Available: https://cloud.google.com/sustainability (visited on 02/20/2023).

[57] Sutton, R. S. and Barto, A. G., *Reinforcement Learning: An Introduction*, Second edition. The MIT Press, 2018, isbn: 978-0-262-03924-6.

[58] "The Istio service mesh," Istio, [Online]. Available: https://istio.io/latest/about/service-mesh/ (visited on 07/11/2023).

[59] Toka, L., Dobreff, G., Fodor, B., and Sonkoly, B., "Machine Learning-Based Scaling Management for Kubernetes Edge Clusters," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 958–972, Mar. 2021, issn: 1932-4537. doi: 10.1109/TNSM.2021.3052837.

[60] "Tools for Monitoring Resources," Kubernetes, [Online]. Available: https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-usage-monitoring/ (visited on 06/04/2023).

[61] Tran, M.-N., Vu, D.-D., and Kim, Y., "A Survey of Autoscaling in Kubernetes," in *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, Jul. 2022, pp. 263–265. doi: 10.1109/ICUFN55119.2022.9829572.

[62] "What are containers?" Google Cloud, [Online]. Available: https://cloud.google.com/learn/what-are-containers (visited on 02/21/2023).

[63] "What is a Container? | Docker." (Nov. 11, 2021), [Online]. Available: https://www.docker.com/resources/what-container/ (visited on 05/29/2023).

[64] "What Is AWS App Mesh? - AWS App Mesh," [Online]. Available: https://docs.aws.amazon.com/app-mesh/latest/userguide/what-is-app-mesh.html (visited on 07/11/2023).

[65] "What is Cloud Computing?" Google Cloud, [Online]. Available: https://cloud.google.com/learn/what-is-cloud-computing (visited on 05/29/2023).

[66] Xu, Y., Yao, J., Jacobsen, H.-A., and Guan, H., "Cost-efficient negotiation over multiple resources with reinforcement learning," in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*, Jun. 2017, pp. 1–6. doi: 10.1109/IWQoS.2017.7969160.

[67] Yan, M., Liang, X., Lu, Z., Wu, J., and Zhang, W., "HANSEL: Adaptive horizontal scaling of microservices using Bi-LSTM," *Applied Soft Computing*, vol. 105, p. 107 216, Jul. 1, 2021, issn: 1568-4946. doi: 10.1016/j.asoc.2021.107216. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494621001393 (visited on 05/16/2023).

[68] Yang, Z., Nguyen, P., Jin, H., and Nahrstedt, K., "MIRAS: Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, Jul. 2019, pp. 122–132. doi: 10.1109/ICDCS.2019.00021.

[69] Yu, G., Chen, P., and Zheng, Z., "Microscaler: Cost-Effective Scaling for Microservice Applications in the Cloud With an Online Learning Approach," *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 1100–1116, Apr. 2022, issn: 2168-7161. doi: 10.1109/TCC.2020.2985352.

[70] Zhang, S., Wu, T., Pan, M., Zhang, C., and Yu, Y., "A-SARSA: A Predictive Container Auto-Scaling Algorithm Based on Reinforcement Learning," in *2020 IEEE International Conference on Web Services (ICWS)*, Oct. 2020, pp. 489–497. doi: 10.1109/ICWS49710.2020.00072.

[71] Zhong, Z., Xu, M., Rodriguez, M. A., Xu, C., and Buyya, R., "Machine Learning-Based Orchestration of Containers: A Taxonomy and Future Directions," *ACM Comput. Surv.*, vol. 54, 10s Sep. 2022, issn: 0360-0300. doi: 10.1145/3510415. [Online]. Available: https://doi.org/10.1145/3510415.

[72] Zhu, C., Han, B., and Zhao, Y., "A bi-metric autoscaling approach for n-tier web applications on kubernetes," *Frontiers of Computer Science*, vol. 16, no. 3, p. 163 101, Sep. 27, 2021, issn: 2095-2236. doi: 10.1007/s11704-021-0118-1. [Online]. Available: https://doi.org/10.1007/s11704-021-0118-1 (visited on 01/31/2023).

Appendix

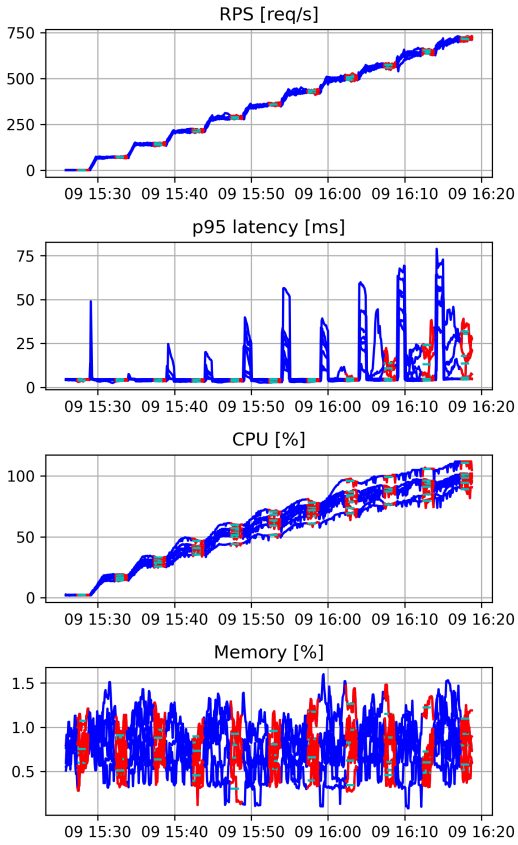


Figure 6.1.1: Measurements on one *productcatalog* replica with increasing constant load.

Figure 6.1.2: Total reward G_T for state $\boldsymbol{S}_2$ on constant load.

Figure 6.1.3: Total reward G_T for state $\boldsymbol{S}_3$ on constant load.

Figure 6.1.4: Total reward G_T for state $\boldsymbol{S}_1$ on step load.

Figure 6.1.5: Total reward G_T for state $\boldsymbol{S}_2$ on step load.

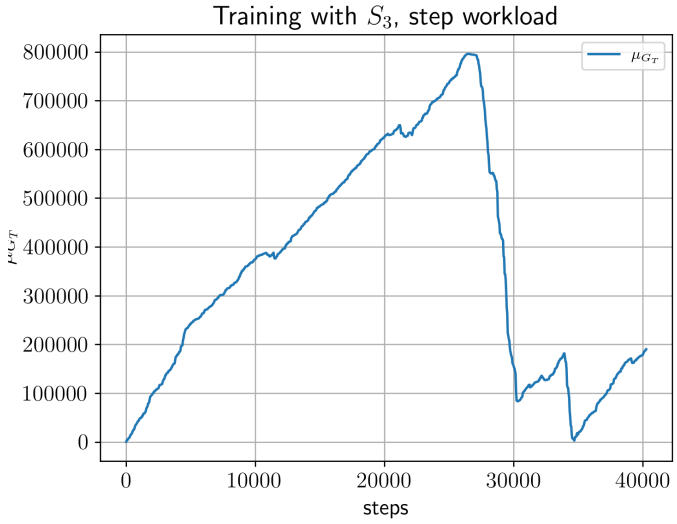


Figure 6.1.6: Total reward G_T for state $\boldsymbol{S}_3$ on step load.

Figure 6.1.7: Total reward G_T for state $\boldsymbol{S}_4$ on step load.

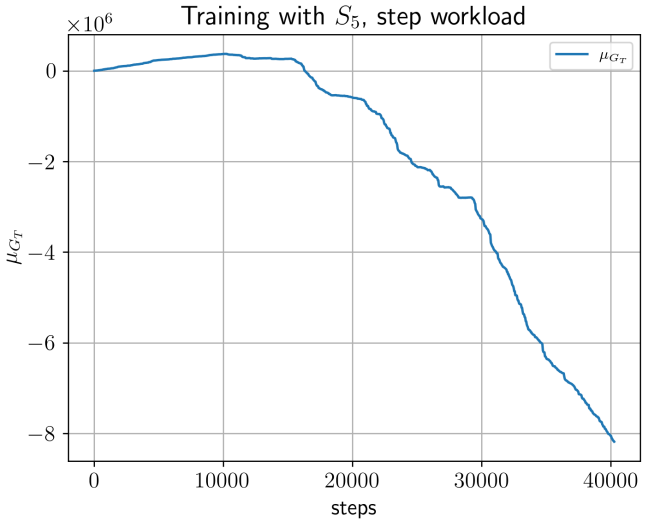


Figure 6.1.8: Total reward G_T for state $\boldsymbol{S}_5$ on step load.

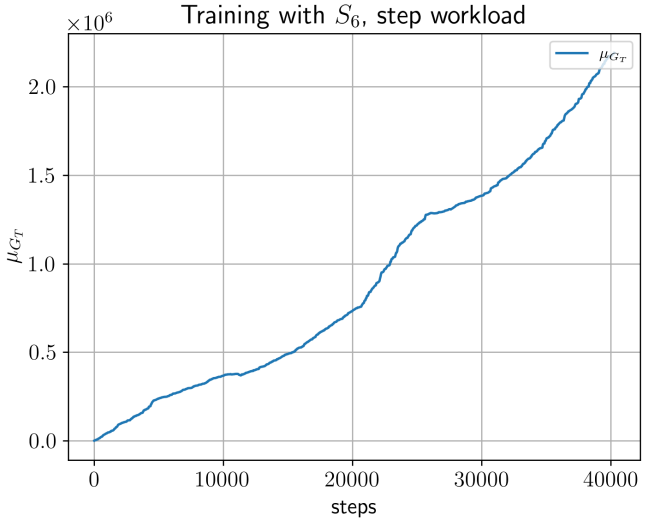


Figure 6.1.9: Total reward G_T for state $\boldsymbol{S}_6$ on step load.

Figure 6.1.10: Total reward G_T for state $\boldsymbol{S}_1$ on periodic load.

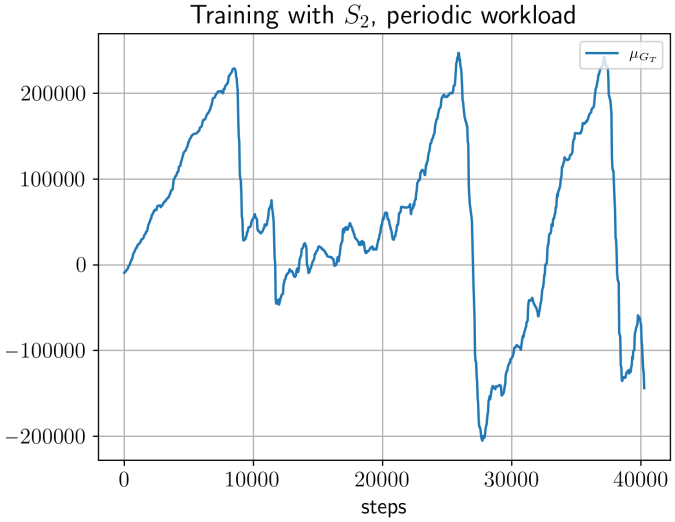


Figure 6.1.11: Total reward G_T for state $\boldsymbol{S}_2$ on periodic load.

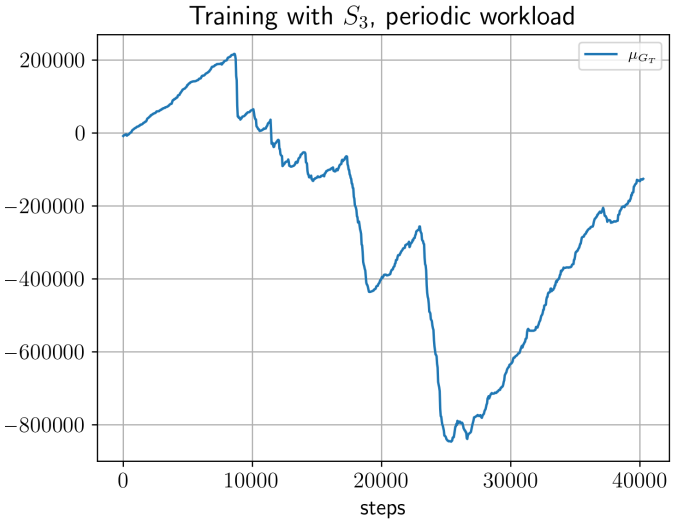


Figure 6.1.12: Total reward G_T for state $\boldsymbol{S}_3$ on periodic load.

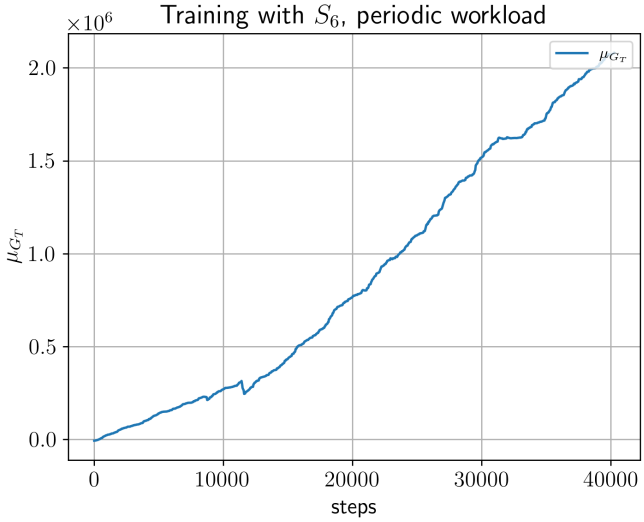


Figure 6.1.13: Total reward G_T for state $\boldsymbol{S}_6$ on periodic load.

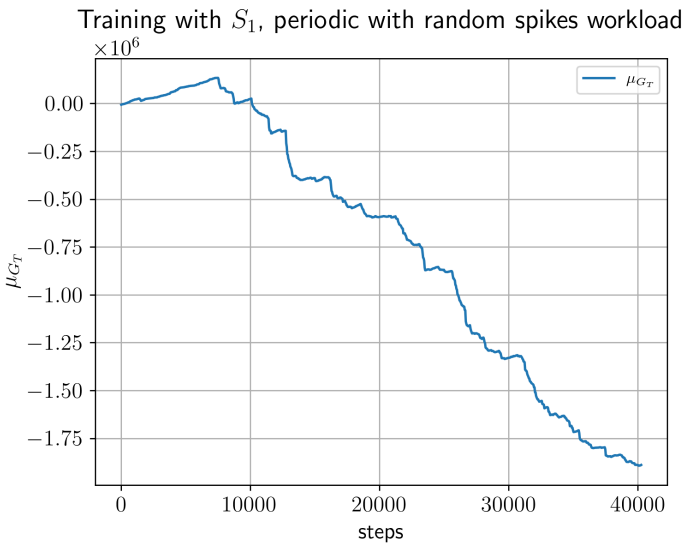


Figure 6.1.14: Total reward G_T for state $\boldsymbol{S}_1$ on periodic with random spikes load.

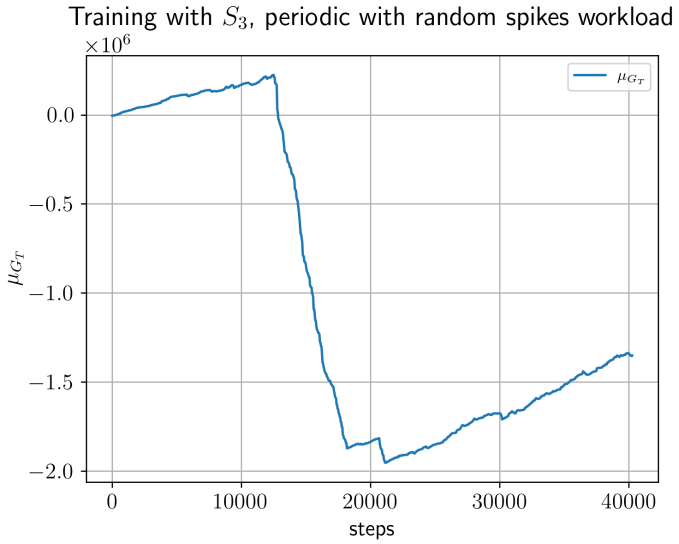


Figure 6.1.15: Total reward G_T for state $\boldsymbol{S}_3$ on periodic with random spikes load.

Figure 6.1.16: Total reward G_T for state $\boldsymbol{S}_5$ on periodic with random spikes load.

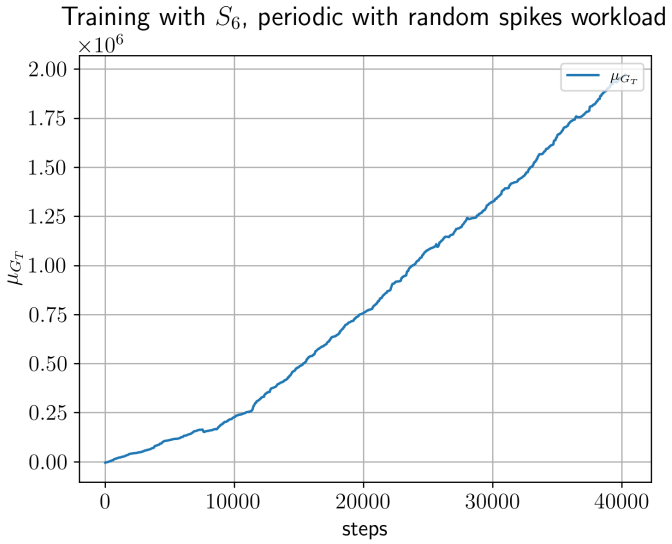


Figure 6.1.17: Total reward G_T for state $\boldsymbol{S}_6$ on periodic with random spikes load.

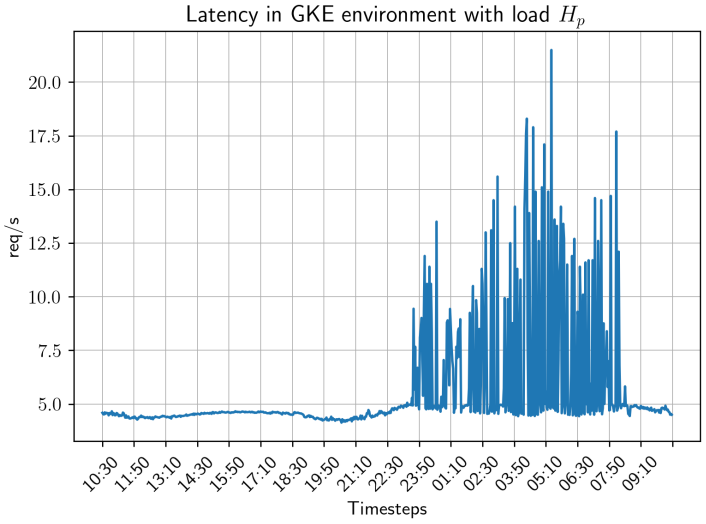
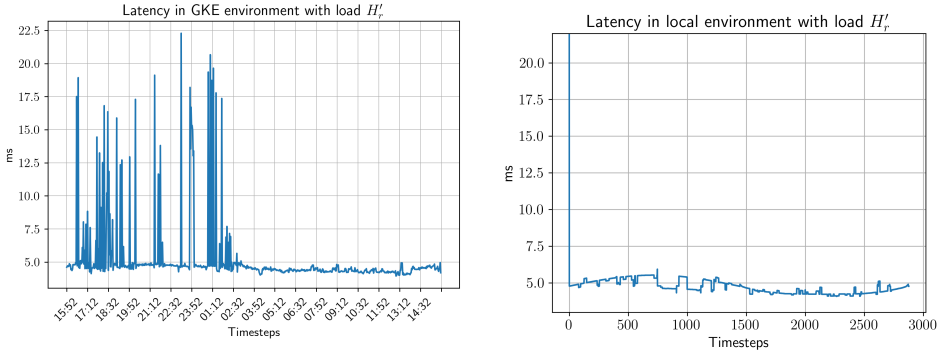


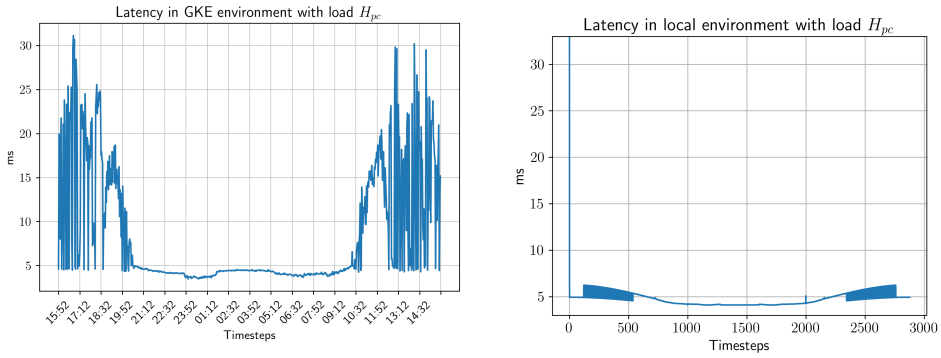
Figure 6.1.18: Latency in GKE environment with load $\hat{\mathbb{H}}_p$ and agent on $\boldsymbol{S}_4$.



(a) Latency from emulated environment.

(b) Latency from simulated environment.

Figure 6.1.19: Comparison on latencies between simulation and emulation with load $\hat{\mathbb{H}}'_r$ and agent on $\boldsymbol{S}_4$.



(a) Latency from emulation environment.

(b) Latency from simulated environment.

Figure 6.1.20: Comparison on latencies between simulation and emulation with load $\hat{\mathbb{H}}_{pc}$ and agent on S_4.

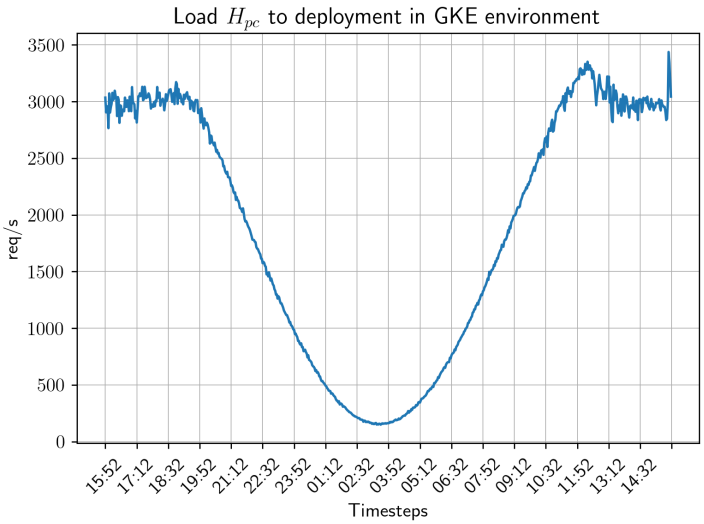


Figure 6.1.21: Load $\hat{\mathbb{H}}_{pc}$ in GKE environment.

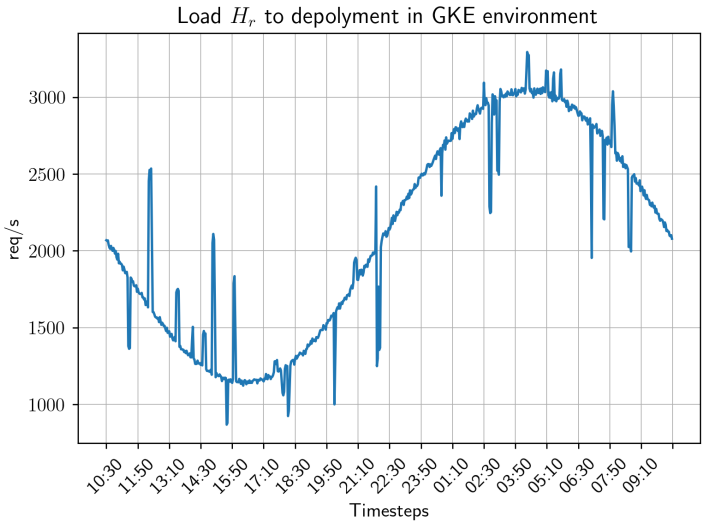


Figure 6.1.22: Load $\hat{\mathbb{H}}_r$ in GKE environment.

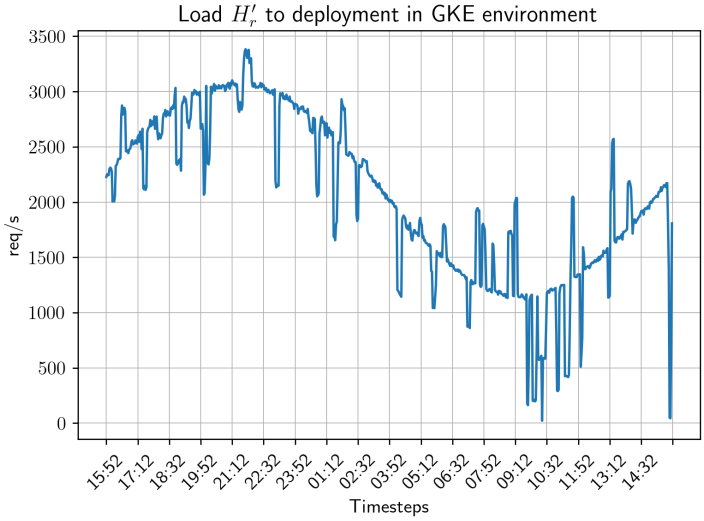


Figure 6.1.23: Load $\hat{\mathbb{H}}'_r$ in GKE environment.

TRITA-EECS-EX-2023:939