POLITECNICO DI TORINO

Corso di Laurea
in Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

Breaking the Challenge of Smart Microservice Autoscaling through Coordination

Supervisors
Prof. Maurizio Morisio
Prof. Vania Marangozova (INP Grenoble)

Candidate
Angelo Gennuso

Anno Accademico 2023-2024

This is the truth of our world. Memories melt with the morning light and then... A new day begins.

Roads stretch out before us. So many paths. Which do you choose? That's up to you.

Sometimes you might run astray. You'll stop, maybe cry in frustration. But you know, that's all right. For the roads they go on without end.

So look up. Face forward, toward your chosen horizon and just...

Walk on.

Summary

In recent years, microservice-based architectures (MSA) have gathered considerable attention for their potential to revolutionize the design and deployment of large-scale applications. This model promotes flexible, loosely coupled, and finely engineered software, making them easier to manage and facilitating DevOps practices.

A critical aspect of MSA is the efficient scaling of microservices while effectively managing resource allocation in the midst of increasing load intensity. Scaling may be applied vertically, by adding more resources to individual microservices, horizontally, by instantiating additional instances of congested microservices, or both at the same time. Recent works have investigated machine learning techniques and, in particular, reinforcement learning (RL) (1) (2), (3) to enhance scaling mechanisms. Taking into account both basic metrics such as CPU and memory usage, as well as higher-level metrics such as end-to-end latencies, these approaches strive to reflect the execution dynamics of microservice applications. However, they predominantly ignore microservice interactions and consider the scaling of a microservice in an isolated manner. In contrast, we propose to exploit information about microservice dependencies and to provide coordinated scaling within groups of microservices.

The way we approach the problem is through the design and usage of a software, called "Grouped Scaler Operator", orchestrating scaling actions among microservice groups, by telling each scaler in a group what to do at a given instant, like scaling in/out or up/down. Our research focuses on the technical feasibility of this task, delving into the world of Kubernetes and its native implementation of controllers, called Operators. One other objective we have is to investigate the applicability and to design a methodology of putting intelligence into coordination, mainly through reinforcement learning, since it can recognise and learn complex patterns present in microservices interactions and then optimise scaling looking at them in such dynamic environments.

To achieve this, various tools have been used, namely Kubernetes (4) for clusters, Kopf Operator Framework (5) for the scaling controller and Autoscalers (6) for scaling. For the metrics, Prometheus (7) and Istio (8) have been used. For the Reinforcement Learning part, PyTorch and various OpenAI API libraries (9)(10) have been used, as long as monitoring tools such as Tensorboard(11) and Weights and Biases (12). The deployments and tests have been carried out on the Grid'5000 national testbed (13) and on a private cluster provided by Orange-EOLAS as a part of the SCALER project (14), in sight of which this project is carried out.

Acknowledgements

This work would have not been possible without the care and the dedication that my supervisor **Prof. Vania Marangozova** and my colleagues in Team ERODS have given me throughout this internship experience. This work is the sublimation of their dedication to me. I sincerely thank all and everyone of them for guiding me here during the professional experience. From the office I thank the friends I have made there, namely **Grégoire**, **Gabriel**, **Ivane**, **Yannick**, **Belen**, **Assan** and also professor **Vania** herself.

I then sincerely thank my close family: my **Mom Giusy** and **Dad Orazio**, my **brother Mattia**, my **two grandmothers Antonietta** and **Concetta**, and my late grandfather **Angelo**, who I miss dearly. Even if in distance and with the gratitude I never gave them, their silent cheer and support gave me a glimpse of hope during the darkest periods.

From my hometown, thanks **Fatima** and **Federica**, two of the friends I care for the most back there. The long talks with them are always refreshing when coming back down there.

Now, a rally of the friends and people that I have met on this journey:

Thanks **Alessandra**, **Fernanda**, **Catia**, **Salvatore**, **Samuele**, **Federico**, **Elisa**, **Riccardo**, especially **Michele G.**, for the moments we spent and the laughs we shared in the first year of University and during the pandemic, and **Sara**, for the love she has shown and for making me grow into a better person.

nThanks **Michele F.**, **Edoardo C.**, **Daniele**, **Francesco**, **Giulia**, **Kaliroi**, **Sebastiano**, **Claudio**, **Farisan**, **Davide F.**, **Daniele DR**, **Giulia**, **Ludovica, Mattia** and **Letizia** for the laughs and hangouts during first year of Masters. In this group, I especially thank: **Alessio C.**, the always petty and funny person whom I vibed with these years; **Simone**, whose wise comments and sarcastic jokes brought me through most of classes and experiences we have had together; **Marco**, for being a soul akin to me and my doomscrolling buddy; **Francesca**, who accompanied me throughout all University and with which I shared a lot of my daily pains about life.

Thanks to **Polisud+BBL**: **Lorenzo**, **Gaia**, **Giuseppe**, **Samuele** again, **Daniele**, **Rosario**, **Martino**, **Danilo**, **Miriana**, **Chiara** and **Claudia**. The moments we have had together were infused with spirit and non-sensical humor. I loved it.

Thanks to the **IEEE HKN Mu Nu Chapter** of the Polytechnic of Turin for having me for one year. The work I have done during the first semester of Masters for them distracted me from the repetitive boring days of lesson. I especially thank **Edoardo** and **Mina**, with which I spent unforgettable days when studying at LABINF in Turin. I also thank from these: **Claudio**, **Alberto**, **Andrea**, **Saro**, **Stefano**, **ManuelManuel**, **Davide A.**, **Elena**, **Francesco**, **Serena**, **Leonardo**, ... and so many other more. I am sorry if you are not mentioned but you were too many!!!

Thanks to the **ERASMUS+ Group**, with which I spent the time of my life in **Grenoble**, and that got me through the darkest of my days. I mainly thank **Linnea** for showing me love and care during the whole experience and more. I also thank **Felix**, for being a guiding light and one of the people which laughs I genuinely cared for; **Michele P.**, for being the always-late but caring friend there; **Alessio M.**, for being my best buddy, one of the best roommates I have ever had and the one that always there for me no matter what; **Luis**, for being the funniest and unhinged and crazy person on the face of Earth, thank you for the care and the laughs; thanks **Ines**, **Sophie** and **Edoardo DB**, the unbreakable trio. Thanks **Nathan** and **Yannick** for the laughs shared together. Thanks **Émile** for being an icon and a good friend. Thanks **Laura** for being a good friend there. Thanks to all the other ones: **Bohdan**, **Davide**, **Gianluca**, **Maxime**, **Johnny**, **Leopold**, **Francesca DM**, **Veronica**, **Stefano**, **Fabio**, **Marianna**, **Hillary**, and probably many

other more that are now lost in the mist of my mind.

If you didn't find yourself in this list, tell me! I have most likely forgot as you're too many :P

All the experiences, all the laughs, all the love you gave me made me grow into another person. I am sad knowing it's all over. But the memories remain, and the will to construct a new life with all those experiences inside of me.

Each and everyone of you, someone more and someone less, contributed to shaping who I very am, in the good and in the bad.

Looking back, I came in at university knowing nothing of the world. There is still a lot to walk for me, but I took big steps and this would have not been possible thanks to the contribution of each one. To put it simply thanks for supporting me in this journey. Thanks for being you when I needed it. Thanks, and just that :)

Thank you.

Contents

List of Tables 11

List of Figures 12

1 Preamble 13
- 1.1 Internship information . . . 13
- 1.2 Scope of the Report and Outline . . . 13

2 Introduction 15
- 2.1 Context . . . 15
- 2.2 Software Architectures . . . 16
  - 2.2.1 Monolithic Architecture . . . 16
  - 2.2.2 Microservice Architectures (MSA) . . . 17
  - 2.2.3 Managing Microservices with Docker and Kubernetes . . . 17
  - 2.2.4 A Microservice application example: TeaStore . . . 18
- 2.3 Scaling . . . 19
  - 2.3.1 Introduction to Scaling . . . 19
  - 2.3.2 What is scaling? . . . 20
  - 2.3.3 About Metrics . . . 21
  - 2.3.4 How to simulate a load: Locust . . . 23
- 2.4 Objectives of Scaling . . . 23
  - 2.4.1 Service-Level-Agreements . . . 24
  - 2.4.2 On dependency . . . 25
  - 2.4.3 Traces . . . 26
- 2.5 State of the Art . . . 27
  - 2.5.1 Introduction . . . 27
  - 2.5.2 State of the Art classification . . . 27
  - 2.5.3 State-of-the-Art Proposals for Scaling . . . 29

3 1st Problem: Grouped Scaling 37
- 3.1 Problem presentation . . . 37
  - 3.1.1 Definition of a Group of Microservices . . . 38
  - 3.1.2 Why grouping microservices? . . . 38
- 3.2 Proposal . . . 39

3.2.1 The Control Loop Pattern and the Operator . . . 40
3.2.2 Grouped Scaler . . . 41
3.3 Operator Solution Implementation . . . 41
3.3.1 KOPF . . . 41
3.3.2 Phases of Running an Operator through KOPF . . . 42
3.3.3 Future for the Grouped Scaler . . . 45
3.4 Experiments . . . 45
3.4.1 Goals of Experiment . . . 45
3.4.2 Analyzed result: Scaling Time . . . 46
3.4.3 Grid'5000 . . . 46
3.4.4 Experiments Setup . . . 47
3.5 Results and Insights . . . 49

4 2nd Problem: RL-based Coordinated scaling 51
4.1 Problem presentation . . . 51
4.2 Reinforcement Learning Introduction and Key Concepts . . . 52
4.2.1 The Agent's Objective . . . 53
4.2.2 A Bit of Classification in RL: Value and Policy Based Methods . . . 54
4.2.3 Agent Model Used: PPO . . . 55
4.2.4 The Concept of Exploration and Exploitation . . . 55
4.2.5 RL model evaluation . . . 56
4.3 Proposal . . . 56
4.3.1 The Multi-Agent Grouped Scaler: Logic . . . 57
4.4 Multi-Agent Scaler Implementation . . . 58
4.4.1 Environment . . . 58
4.4.2 Agents . . . 62
4.4.3 Workload Generator . . . 62
4.4.4 Monitoring . . . 62
4.5 Experiments . . . 63
4.5.1 External configurations: Nodes and Groups . . . 63
4.5.2 Internal Configurations: Hyperparameters . . . 63
4.5.3 Runs and Evaluation . . . 65

5 Conclusion 71

List of Tables

4.1 Hyperparameters for PPO Training 64
4.2 Environmental and Training Parameters 64
4.3 Summary of Experiments 65

List of Figures

2.1 Docker and Kubernetes 17
2.2 Monolith vs Microservices Approaches. Picture taken from Atlassian 18
2.3 TeaStore benchmark 19
2.4 Prometheus 22
2.5 Locust 23
2.6 Istio 25
2.7 Example of a Service Dependency Graph 26
3.1 Dependency Graph for TeaStore and DeathStarBench's Social Network 39
3.2 Resource Usage from the TeaStore Application under a light stress 40
3.3 Grouped Scaler Operator schema 45
3.4 Grid5000 46
3.5 First 3 experiments times with 2 and 3 worker nodes 47
3.6 4th experiment times for the various groups with 2 and 3 worker nodes 48
4.1 Reinforcement Learning Schema 53
4.2 Actor-Critic schema 55
4.3 MARL Grouped Scaling schema 58
4.4 Tensorboard and WandB 62
4.5 Rewards for the various runs throughout all the steps 66
4.6 Training Metrics among Agents 68
4.7 States of run E. per Group 69
4.8 Difference of behaviours between Agents and respective hypothetical HPA best replicas 70

Chapter 1

Preamble

1.1 Internship information

In this report, I am going to write about my **M2 Internship**, in which I participated from the **7th February 2024** to the **3rd July 2024**. It took place in the Laboratoire Informatique Grenoble (LIG), which is a warm and friendly environment, full of interns, PHDs and professionals ready to help for anything at any moment. I was assigned to the ERODS team (15), which is a research team associated with CNRS, Grenoble INP and UGA. The branch that the team covers is summarized Cloud and Distributed Systems, so the topic of this report will be on that. Software and Cloud terminologies and tools will be therefore presented and used throughout this dissertation.

I worked on the SCALER (14) project (Smart **SCAL**ing for Micros**E**rvice A**R**chitectures). It is an **ANR-funded** project, led in coordination by various entities, namely UGA, INRIA, Orange Innovation and Orange EOLAS.

1.2 Scope of the Report and Outline

The main motivation of this work is presenting a solution on the question of **Grouped Microservice Scaling**.

Microservices-based applications are the trending solution in the market of distributed computing. Many companies like Netflix or Meta exploit them to provide their numerous services. The scaling of the microservices in those applications is therefore crucial for the correct assignment of resources and guaranteeing a good service to the end-user. In this work, the question of **Grouped Scaling** is presented: can we group microservices and scale them altogether? What are the technical means to do it? Does it bring any enhancements to the already pre-existing solutions? What can be taken from the State of the Art solutions and applied to our solution?

The report will go through some presentations about the topics in question in Chapter 2. The whole context will be presented, what scaling is in detail and why it is needed. Finally, some State of the Art solutions about Scaling will be presented and what they achieved, along with some ideas and concepts that are later reused in the work.

Chapter 3 covers the first two questions presenting the problem, while also formally presenting the **Grouped Scaling solution** to the problem in Section 3.2: a prototype tool called

Grouped Scaler Operator has been developed, which has been used to study the context and if this task is technically feasible. Chapter 4 covers the two other questions: Inspiration from (16),(2),(3) is taken and applied to our context, trying to give *intelligence* to our grouped scaling mechanism with the aid of **reinforcement learning**. Here then, theoretical notions about it will be presented in 4.2 and a subsequent solution is presented in 4.3.

In Section 3.4 and 4.5 the results of these two solutions are presented. Their goal is to mainly answer the question that this mechanism works, and then respectively present some insights and consequences of scaling more microservices together in the first one, while presenting some promising results for the second one.

Chapter 2

Introduction

In this chapter, a first look at the context of Microservices is given in 2, continuing with explaining what scaling is in 2.3, the used metrics in 2.3.3 and what microservice dependencies are in 2.4, as also presenting the standard-de-facto technical tools used to cover each part and that are therefore used in this report. In 2.5 a look at the State of the Art on scaling is finally given.

2.1 Context

The field of *Software Engineering* has been constantly evolving since its beginning, always eager to develop methods to design software in ways that most suit the context on which they are applied to.

In the beginning, every coding language was fairly simple and for the most low-level, without much abstraction. Every margin of improvement was visible by improving the data structures, algorithms, compilers used on the projects. The product would then have been used as a whole in the end, after being first being linked with the libraries and then compiled and used in one piece (i.e. in an executable file).

Since the introduction of *Object-Oriented-Paradigm* (**OOP**), almost every aspect of *Software Engineering* and therefore development has tendentially evolved from developing every part of the product all in one place to be developed in a **modular** fashion, trying to break down the components of every single problem into smaller pieces with their defined and isolated functionalities. These pieces, that we can adequately call **modules**, can therefore be assembled to create a fully-functioning product, each of them containing a well-defined logic separated from the ones of other pieces. These modules can communicate with each other using calls to their respective exposed interfaces. From this point of view, one could create a fully functioning application by attaching various pre-compiled pieces that communicate with each other while still being independent.

This is the rough idea behind *Microservices*. But let us start from the beginning by formalizing some concepts.

2.2 Software Architectures

The possibility to split the process into modules calls the need to define the higher-level/abstract way of defining the possible relationships between these modules, and how that can be used to define ways to reason about the performance, scalability and scope of the products, while also giving a powerful tool to enact techniques of Project Management to the process (useful in big companies and research group that want to better manage their ongoing projects). This need is met by introducing the concept of *Software Architectures*.

The *Software Architecture* of a system represents the design decisions related to overall system structure and behavior. Architecture helps stakeholders understand and analyze how the system will achieve essential qualities such as throughput, availability and security. (17)

From the definition it is possible to note that basically every aspect of the software functionality is related to this. This also implies how the code is structured, how the modules interact with each other, if they are completely separated or there are parts that run together.

A first example of a naive and simple type of architecture that will be described now will be the **Monolithic Architecture.**

The following definitions have been inspired by (18).

2.2.1 Monolithic Architecture

Let us start with an example: a simple application that requires functionalities like *Authorization*, a *Database*, a *Front-end*... is to be created. How do we design it?

The first naive solution would be for various developers to create pieces of software that are compiled and run together at the same time, by dividing into teams of expertise (a team on the *Front-end*, a team on the *Database*...). Therefore, the most naive way of looking at this is creating an application in which all of its pieces are encapsulated into one application and all of them run together. The performance would be determined by the systems' own capacities to process data/computations and latencies between them and the external sources of data/calls.

A Software that strictly follow this architectural pattern is categorized as having a **Monolithic Architecture**, as the fully-functional applications are defined as **Monoliths**. Using this architecture, each module of the application communicates with another one internally, without using the network.

It could be seen as the easiest way to develop an application and it is generally easy to create prototypes with it. When the size of the project scales up it is definitely harder to keep using this kind of architecture. Every part of the software requires different expertise that can only be met by teams of people with different skills. From then on, it will also require maintenance. It also goes without saying that creating a single application with these specifics creates a natural single point of failure: if a part crashes, everything could potentially crash.

If the application throttles, giving it more resources is also hard: the whole application will need to scale and no distinction is made between all the various components. If the application is big, scaling it in "one-block" might not be optimal.

2.2.2 Microservice Architectures (MSA)

Following the aforementioned tendency of modularizing of processes and software, a natural evolution of the monolithic architecture is the one defined as *Microservice Architecture*, **MSA** for short.

Using again the same example as before, this time a more refined solution would be using the principle of the separation of concerns. As before, some developers would focus on the *Front-End*, some on the *Database* part and so on. This time they all produce different fully-functional modules, that from now they will be defined as **microservices**. Their different pieces of software would be independent from each other (to a certain extent, on which we will will focus later in 2.4.2), therefore easier to maintain on a larger scale, on a level of Project Management and on a level of future maintenance.

For the application to work, these services will still have to communicate between each other and eventually with the outside world. This time, the network is used. At the cost of possible internal latency given by the network, the services will now communicate each other using **API calls**.

This gives us an application that is made of easily-deployable **microservices** that can communicate with each other, easier to debug and more fault-tolerant (i.e. some of them crash and other microservice with the same functionality would eventually absorb the eventual load, maintaining the application functionality and availability at an eventual performance loss cost).

It is by far nowadays the most used paradigm in the world of *Cloud Computing*, as the **natural independence of microservices** in terms of **deployability** permits for them to be **distributed** over a large number of nodes and clusters, while still maintaining their single functionalities in the whole context of the developed application.

A graphical comparison between Monolithic and Microservices Architectures is given in Figure 2.2.

2.2.3 Managing Microservices with Docker and Kubernetes

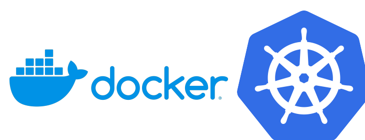


Figure 2.1: Docker and Kubernetes

Modern application deployment is heavily reliant on **containers**, which bundle code and its dependencies to ensure consistent execution across diverse environments (19). Docker(20) is a prominent tool for creating and managing such containers, where developers define containers using Docker Images, built from instructions in *Dockerfiles*. A **Dockerfile** typically includes the base operating system, required packages, and the application to be executed, ensuring reliable and reproducible builds. These Docker Images can then be uploaded to online registries like Docker Hub for sharing.

However, deploying microservice-based applications, which consist of multiple interdependent services, requires more than just containers. This is where an **orchestrator** becomes essential. Kubernetes(4)(**K8S**) is the leading platform for managing microservices in cloud environments . Kubernetes encapsulates containers into deployable units called ***Pods*** (21), which are managed through the Kubernetes API. Kubernetes environments typically consist of multiple computational nodes, with at least one serving as the **controller** and the others as **worker nodes** hosting the applications. Kubernetes uses **.YAML** files to define and manage microservices, specifying the source of container images (from online registries like Docker Hub or local registries) and interacting with the *Control Node*. These files **usually** also define **resource limits** (such as CPU and memory), which define how many specific resources can a single microservice replica attain from the cluster. This will prove fundamental in scaling.

When deploying microservices on Kubernetes, each microservice is represented by a **Deployment** object. The **Deployment** object acts as a blueprint, containing all the necessary configurations and environment variables for the application. However, it is important to note that the Deployment object itself does not represent the microservice instance but rather the model or template that Kubernetes will use to create instances within Pods.

Once a Deployment is created, Kubernetes extracts the Docker image specified in the **.YAML** configuration, creating instances of the microservice within Containers that are then placed in Pods. The Kubernetes API manages the pods in terms of their aliveness and health, while scaling configurations are managed at the Deployment level. When a scaling action is initiated, the Kubernetes cluster updates the configuration settings, and the **Controller**, which oversees the pods, enacts the scaling action accordingly.

Kubernetes is inherently designed for distributed deployment, ensuring that computational workloads are balanced across the cluster nodes. This distributed approach is crucial for efficient scaling and resource utilization, and it will be further explored in subsequent chapters.

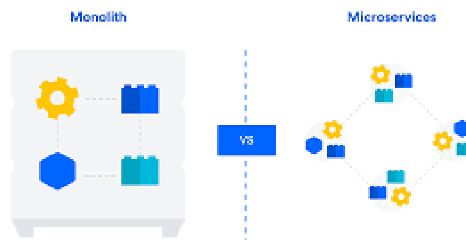


Figure 2.2: Monolith vs Microservices Approaches. Picture taken from Atlassian

2.2.4 A Microservice application example: TeaStore

For the sake of this dissertation, the example application **TeaStore**(22) is now presented, as used in both contributions given in this work, as it also represents an easy introductory example of how a classic microservice application can look like.

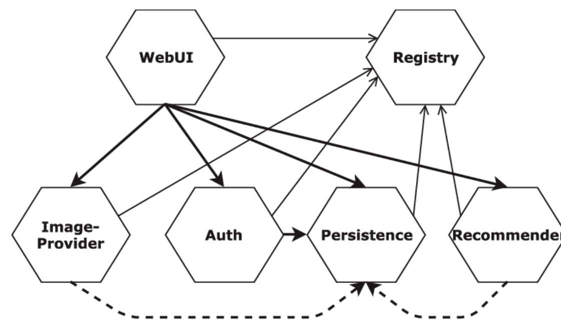


Figure 2.3: TeaStore benchmark

The TeaStore application benchmark is composed of 5 microservices, 1 database and 1 registry. The 6 microservices are:

- **teastore-webui**: the one on which users connect.
- **teastore-image**: the one providing the image files to the webui for the teas to be displayed on the website.
- **teastore-auth**: the one tasked with managing log-ins and log-outs.
- **teastore-persistence**: the one that keeps a local history of the database.
- **teastore-recommender**: the one that produces recommendations for products and items in the cart. The **teastore-registry** is the one instance which all microservices communicate with. It is responsible for load balancing and other features, but will not be considered as a microservice together with **teastore-db**.

In the **.YAML** file of the deployment of the application, various configurations can be deployed. One interesting pararemeter are the resource limits on resources. For the CPU, all microservices have been restricted to 2 **cores per second**, while for Memory to 10 **GiB**. If not specified, this will be the standard value for these parameters.

2.3 Scaling

2.3.1 Introduction to Scaling

In the context of *Cloud-Oriented Applications*, where everything is developed with the objective of independence and distribution, it is essential for the parts of applications to be able to receive data, schedule its processing and then process it, optionally giving back a result and eventually storing it. In an ideal situation, all data should be processed correctly, which can be translated in mathematical terms by defining a term called **Throughput** $\in [0,1]$. A throughput equal (or close) to **1** means that the system can handle (almost) all data and process it correctly; vice versa, a system with a throughput equal (or close to **0** means that the system can handle (almost) no data at all, being unable to produce enough results.

As ideal as this sounds, the perfect throughput cannot be reached. This is due to various problems like: network problems, internal crashes and the systems' incapacity to handle big amounts of load. This last one means that some parts of the application could potentially crash because too many users try to make requests compared to how much the system can actually handle.

To guarantee a high throughput on our applications, *Scaling* is needed and applied. The mechanism of scaling will be now presented and explained, so to what problem it answers, what mechanism it uses and what it achieves.

2.3.2 What is scaling?

Generally speaking, **Scaling** is an action taken on an application that gives it the ability to function or perform better as a function of a change on the load on it. Scaling takes some **metrics** as **inputs**, **applies** a **policy** to scale based on calculations based on those and **performs** a **scaling action** as output.

It is possible to go further and define some more types of scaling. In the context of *Cloud-Oriented applications*, the two easiest types of scaling that can be applied are: **Vertical** and **Horizontal**. Other kinds are also present, but require further knowledge that will not be presented here. It is important to note that scaling is applied per microservice. This means that scaling actions can be different between different kinds of microservices, based on different needs. This separation is indeed positive, as hinted at in 2.2.2, because it can lead to scaling microservices that only need it, instead of scaling the whole application, potentially using less resources.

As general as the next section can be, *Docker and Kubernetes* terminologies will now be used, as that will be the main platform on which we will operate.

Vertical Scaling

In Kubernetes, applications are managed as pods, with resources such as CPU, memory, and disk allocated based on .YAML configurations. Without resource limits, microservices can consume all available resources on a node. Setting limits ensures controlled resource allocation, preventing resource contention across applications.

When resource limits are too low, **Vertical Scaling** can be used to adjust them. **Scaling Up** increases resource allocation to a microservice, addressing potential performance issues. Conversely, **Scaling Down** reduces resource limits, freeing resources for other services. These adjustments are typically straightforward and involve updating configuration limits, meaning that scaling takes virtually no time: the application will take the new resources, and with time the effect should be seen.

In Kubernetes, **Vertical Pod Autoscalers** (VPA) automate vertical scaling by monitoring resource usage and adjusting limits based on predefined thresholds (23), performing a scaling action at each timestep. However, this approach is less common in production and research because simply adding more resources may not improve performance in a microservice architecture. Sequential processing limitations still persist when the workload increases, as multiple users interacting with the same instance of a microservice still generate a queue of request handling, meaning that vertical scaling might actually not help at all in this case.

Horizontal Scaling

As with vertical scaling, resources like CPU, Memory, and Disk are allocated with defined limits. Another solution for performance issues is **Horizontal Scaling**. **Scaling Out** creates additional **instances/containers**, or **Replicas**, of microservices, distributing the load and enhancing performance through parallel processing. Conversely, **Scaling In** reduces the number of replicas, freeing resources when demand decreases.

Horizontal scaling takes longer, depending on the cluster's capacity and workload. The Kubernetes controller balances the scaling tasks, ensuring smooth replica creation or termination.

In Kubernetes, **Horizontal Pod Autoscalers** (HPA) automate horizontal scaling by monitoring metrics and adjusting the number of replicas at each timestep based on the following heuristic formula:

$$replicas_{desired} = ceil(replicas_{current} * \frac{metric_value_{current}}{metric_value_{desired}}) \tag{2.1}$$

If multiple metrics are used, the maximum value determines the scaling action.

Horizontal scaling is increasingly utilized in both production and research environments, as it allows for the distribution of replicas across nodes, thereby enabling more efficient handling of API requests to the application. This approach is also gaining traction with the exploration of *Machine Learning* techniques, which aim to enhance traditional heuristic methods, as seen in (24)(1) and also discussed later in 2.5.

2.3.3 About Metrics

As said at the beginning of 2.3.2 in **horizontal** and **vertical** scaling, metrics are the data used to compute the scaling action. In the context of microservices, metrics are time-based data that report on specific aspects of their lifecycle. Typical metrics that can be seen (and will be used here) are **CPU** and **Memory Utilization**, as well as application latency.

Resources Utilization

At each timestep t, each microservice has an associated number of replicas and specified values for CPU and Memory limits. Each replica is containerized and managed in a Kubernetes pod. Each pod consumes those type of resources. For CPU the measurement unit is **cores-per-second**, for Memory the measurement unit is the total amount of **allocated bytes**. Each pod has also associated limits to the usage of those resources. Per pod, we define the **relative Resource Utilization** (**RRU**) as the ratio between the absolute usage (**RAU**) and the corresponding limit (**RLT**):

$$RRU_{pod} = \frac{RAU_{pod}}{RLT_{pod}} \tag{2.2}$$

This represents how much each pod is using its assigned resources, effectively representing eventual under-usage or throttling.

$$RRU_{microservice} = avg_{microservice}(RRU_{pod}), \forall pod \in microservice \tag{2.3}$$

Equation 2.3 represents how much each microservice is effectively using the assigned resources, and therefore providing valuable information on scaling. $RRU_{microservice}$ is in fact the $metric_value_{current}$ used in 2.3.2 for horizontal scaling, whereas $metric_value_{desired}$ is its **desired value** that the scaling action wants to achieve. RRU_{pod} and $RRU_{microservice}$ are **usually** $\in [0,1]$, but spikes and throttling make the upper limit a soft one, meaning that they tendentially stay in that range, but at a given timestep t it can exceed the limits. This is due to Kubernetes own ways of managing the limits and the resources usages among pods, so we will not delve too much into technicalities. We will just say that values over 1 effectively mean, in this context, an overusage.

How to retrieve the metrics per timestep t in Kubernetes?

Gathering metrics on Kubernetes: Prometheus

Figure 2.4: Prometheus

The VPA and the HPA require an extension called **Metrics Server**(25). We will not use it as our objective is to collect and exploit more metrics than the ones that this tool exposes. For that, we use Prometheus.

Prometheus(7) is a **metric monitoring system** that collects metrics from attached systems, stores and eventually displays them in a time-series database. This tool is **open-source** and easy to deploy in Kubernetes clusters, and brings a numerous amount of features with it, including its own *Query Language*, **PromQL**. It is easily deployable thanks to the many installation packages available on GitHub. The one that is used in this work is the **kube-prometheus-stack**(26), which is a comprehensive and configurable way of managing a Prometheus configuration and subsequently its deployment.

Prometheus uses a series of scrapers and API servers in the Kubernetes cluster to scrape the metrics from all the individual entities that expose such metrics, takes them at regular intervals and aggregates them to form a wide variety of different metrics. In our case what we need is pod/containers aggregated metrics, and these easily provided by standard configuration of Kubernetes pods that mount the **cAdvisor**(27) performance metric collector. The metrics are usually collected and updated in the order of **seconds**, as bigger time-frames lack the possibility of seeing spikes and problems, whereas smaller ones would require a lot of API calls in the cluster, possibly overloading it.

Through the usage of PromQL, a client (i.e. a Python script) can query Prometheus' endpoints and periodically request certain metrics that satisfy certain fields queries (like the ones associated to a certain application). This will prove useful later in 4.3.

2.3.4 How to simulate a load: Locust

Figure 2.5: Locust

For the sake of knowledge, another tool will now be presented and given for granted in next sections.

As a base level, to gather metrics just an application and Prometheus are needed. But the metrics would be trivial as the microservice is just running but no one is using it. In this way, the real capabilities of the application and of the cluster cannot be tested.

In development environment, usually on microservices based application, it is not unusual to rely on specific softwares that simulate a variegated amount of load on the application. These tools usually rely on initializing various clients that fake user interaction by "using" the application as one real user would. In this way, all the various aspects of the application workload absorption can be inspected, also thanks to metric monitoring tools like Prometheus.

A popular tool to generate such kinds of workloads (which are the ones that interest us the most being in a Cloud environment) is Locust (28), which is open-source.

Locust usually takes as input a file named **locustfile** and generates users that will follow the request schema defined in this file. The Locust client will have to manage the connection to the specific URLs, the workload parameters and so on.

2.4 Objectives of Scaling

As seen from 2.3.2, the two most basic types of scaling are applied in Kubernetes Clusters through the usage of objects called **Autoscalers**. These autoscalers are objects created with one objective: get attached to microservices, monitor them continuously and scale them according to some policy. Their objectives then are:

- Keep the application **alive** and **functioning**;
- Keep an **optimal usage** of the **resources** thanks to the defined desired thresholds over which the microservices are scaled;
- **Optimize** the total **resources usage** of the cluster;
- **Distribute** instances in the **Cloud**;
- **Guarantee** Service Level Agreements (**SLA**).

These points are generally touched in the State of the Art that will be presented later. A quick digression on the last point will now be made.

2.4.1 Service-Level-Agreements

Some methods are studied to guarantee the respect of Service-Level-Agreements (**SLA**, shortened), which are **service metrics goals** defined on a contractual level between the stakeholder and the developer who is producing the software or, more generally, between a producer and a customer. They generally state which kind of performance is expected.

Latencies and Quantiles

An appropriate example of this is **Latency**. For example, A customer might not want an application to generally present an end-to-end latency (i.e. response time in milliseconds) that is over $5ms$. These kinds of metrics are defined then at an application level, so in the context of this work, for each timestep t it will be defined as a real value that is associated with specific quantiles of latency distribution.

Quantiles are crucial in understanding and guaranteeing the performance of a microservice application. The most commonly used quantiles in Service-Level Agreements (SLAs) are the 0.50, 0.95, and 0.99 quantiles, which help characterize the distribution of latencies experienced by users.

- **0.50 Quantile (Median)**: This represents the median latency, meaning 50% of the requests will have a latency below this value. It represents the **"typical" latency** that users experience. It is useful for understanding median tendency, but it does not reflect the performance at the extremes.

- **0.95 Quantile (95th Percentile)**: This indicates that 95% of the requests will have a latency below this value. The 0.95 quantile is critical for understanding the "**tail latency**," which affects a significant minority of users. It helps in ensuring that the majority of users are not experiencing unacceptably high latencies.

- **0.99 Quantile (99th Percentile)**: This represents the latency below which 99% of requests fall. The 0.99 quantile is often used to identify **outliers** and ensure that even the most extreme cases are within acceptable limits. In highly responsive systems, ensuring that the 0.99 quantile is **within** the SLA limits is crucial for maintaining a high-quality user experience.

These quantiles allow developers and operators to design and monitor systems that meet performance expectations under various conditions, ensuring that the service level agreements are met not just on average, but also for nearly all users.

Since an SLA objective can be defined for each one of these quantiles, we can define the metric of the SLA Preservance Ratio at timestep t (SP_t) as seen in Equation 2.4.

$$SP_t = \begin{cases} \frac{SLA_QuantileLatencyX_{limit}}{QuantileLatencyX_t} & \text{if } QuantileLatencyX_t < SLA_QuantileLatencyX_{limit}, \\ 1 & \text{otherwise} \end{cases} \tag{2.4}$$

Gathering metrics about SLA: Istio

Figure 2.6: Istio

To gather informations about microservices, the **Kubernetes + Prometheus** combination might not be sufficient in some situations. To account for that we use a Kubernetes add-on: Istio(8). Istio brings a lot of features to the microservices environments. The feature that is most relevant for us the most in this case is the addition of metrics that particularly measure the quality of the microservice communication between each other, i.e. latencies. From Istio, with a special metric, we retrieve the information about the whole microservices latencies that translates to the latency that the final user could eventually be subject to. Retrieving this metrics is possible thanks to the integration of Istio with Prometheus.

This proves particularly useful in retrieving information about SLA. This this also will prove useful later in 4.3.

2.4.2 On dependency

As mentioned at the end of 2.3.2, the **scaling** action is **applied** individually **per microservice**. Using objects like HPAs, this would mean that at each given "turn" of decision, for different microservices HPAs can decide to scale to different number of replicas based on pods' individual metrics and how the autoscalers are set up. This implies that, in our point of view, in most of the cases, scaling would be applied just considering the microservices independently. This is enforced by the metrics used to scale, which have been explained in 2.3.2 for **horizontal scaling** and in 2.3.3 Are microservices then really independent?

As explained before, an application following a microservice architecture still implies that the microservices communicate with each other and that some microservices need other to function (properly). As example, we take an application that is composed of a *WebUI* microservice, a containerized database and an *Authorization* microservice. The *WebUI* needs data to load from the database, hence we can derive a **first dependency**: the *WebUI* still generally works, but will show no data because the database is not working. Without the *Authorization* microservice working correctly, the *WebUI* would not be able to manage that part correctly and so on. Still the *WebUI* can potentially work, but the whole application doesn't. This and other examples of dependencies expose a more precise definition of independence: Microservices are independent in their **deployability** and **scalability**, but are subject to other microservices' changes in performance or crashes.

As a microservice of an application starts to get potentially more and more load, other microservices can start to use more resources, because required to work from the original one.

This does not necessarily imply that all the microservices will undergo the same resource utilization (i.e. a database will use a lot less CPU than a *WebUI*), hence independence on scaling is applicable and justified.

A way to analyze these dependency behaviors is to collect artifacts called Traces and analyze those, thanks to tools like Jaeger (29).

2.4.3 Traces

Microservices in an application communicate through API calls. If we see microservices as nodes and calls as directed edges in a graph, we can construct a type of graph that can well visually describe how various services communicate with each other and which microservices depend from certain other ones. This graph can be called **Service Dependency Graph**. It can be either be constructed *a priori* when defining the architecture of the software or *a posteriori* by in fact collecting traces. This graph is shown in the example of the TeaStore example application(22), shown in Figure 2.7

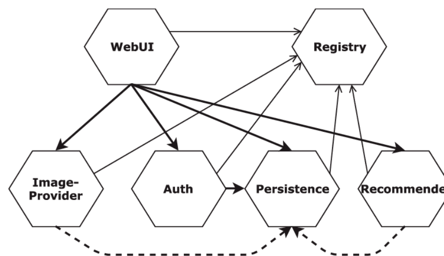


Figure 2.7: Example of a Service Dependency Graph

When a request is delivered to a microservice, this is processed by said instance and then other microservices will be called in a hierarchical structure; a **Trace** can be defined as the data that we can collect for a request from its beginning microservice to the ending microservice(s). These tools give more insight into how a certain application works and how it can be readjusted to work better under certain conditions. Studying traces and internal applications latencies can also give powerful insights about how microservices are connected between each other and how a little difference in one can have a big impact on others. Interesting examples of traces are found in (30), in which the authors study a large amount of data available on the Alibaba cluster, effectively aiding research on the study of this matter.

Dependencies are then shown to be a crucial factor in microservices design: should scaling also account for that? In our opinion yes. For that we propose a possible mechanism to approach that in 3.2.

We put this topic now to a hold and: the next section will in fact present how the research community approached the scaling problem and the solutions various teams provided for the study.

2.5 State of the Art

2.5.1 Introduction

In the context of research for **scaling**, the main tendency is to look at different and innovative ways to scale the microservices. Usually, the standard heuristic scalers are taken as a base and then compared with to see if the new solutions are generally better in terms of general performance and costs. The various strategies we are going to see address scaling using different ideas, but sharing the common goals listed in 2.4.

The final result is mostly the same throughout: every method presents an idea that is always better than Kubernetes autoscalers but that is hard to compare with other solutions in the same contexts. This is because this problem is very specific on the application it is being tried on, the clusters and the natural randomness of execution. We will see some state of the art solutions with their strong and weak points.

The state of the art research has been aided first by (24) and subsequently by (1), and then finalized by personal research. These two reviews take into account all the work from the beginning until 2022. They cover all the concepts extensively and to this day no better surveys or reviews were found. The following classification is adapted from (24).

2.5.2 State of the Art classification

Today, the main techniques encountered are three:

- **Threshold-Based Rules**: Impose a threshold based on more or less complicated metrics and then scale based on the outcome of the function. The most similar approach to already pre-existing autoscalers.

  Pros:

  - These rules are **easier** to **understand**: the scaling decision is explained by a mathematical formula, that can get better with adequate parameter tuning;
  - **Easier** to **reproduce** since no Machine Learning or Reinforcement Learning are involved.

  Cons:

  - The approach is **only reactive**: it means that no prediction is made on the workload and so no action is taken in hindsight, as it is just taken to solve a problem on the instant. It makes this the weakest approach since this implies more problems, mainly like delay in the resource allocation (since horizontal scaling takes time) and cold initiation of service instances(24).

- **Machine-Learning**: It is exploited to analyze workloads or specific application conditions and pass the subsequent result to an algorithm that takes that as input and scale accordingly. With the advent of **Neural Networks** and **Deep Learning**, many new opportunities arose, thanks also to the use of encodings (like in (3)) or Bi-LSTM (like in (16)).

  Pros:

- The more variegated the datasets and possible situations are, the more **expressive** the models can be, thus definitely giving better results;
- This approach is **proactive**: it can be leveraged to analyse through statistical models (ARIMA for example) and predict the possible evolution of workloads and application states, thus giving more information for a possible **predictive scaling**.

Cons:

- With Machine Learning we to rely on probabilistic and statistical models: this means that models can still give sub-optimal results and predictions if the proposed input has never been seen or too different from the training dataset used;
- Another problem with these models is that they have to be evaluated with test datasets and usually cannot be interpreted , they being abstract by nature, thus requiring a big effort in total to gather an expressive enough training dataset and a complete test dataset for the wanted analysis.

• **Reinforcement-Learning**: Solutions of this kind in scaling are of a different species from the other two. This technique, of which we will talk extensively in the second part of this report 4, is used to try-and-learn various situations, decide on an scaling action based on the presented states of the application, get numerical rewards for what happened after that and learn. This approach is definitely the most studied in this context.

Pros:

- It can potentially **learn** various situations the application can go through in all its life, exploiting the learnt knowledge to act on similar situations;
- It is live: the model training is done on the live cluster, so there is no need to generate big datasets beforehand. It is important to try and simulate all possible combinations of the application states to make the learning effective.

Cons:

- It is highly sensitive to the states and the reward function that is chosen (more on that in 4.4.1);
- It is a problem with a lot of hyperparameters. Tuning them is a problem of its own, since it is highly dependent on the whole environment, states, and rewards;
- Can potentially not converge to an optimal global solution, but to a optimal local solution that will not solve the problem any better than a threshold-based scaler;
- Training times can be really high if the neural networks are big.

These methods have been presented individually, but the presented research is mostly presenting a mixture of both, particularly with Machine Learning and Reinforcement Learning, even if they can potentially present a lot of problems. Their results have been positive, so it is natural that research tends to that.

The first types of solution are the least present, as the state-of-the-art is transitioning toward the other two solutions, since the prediction of workload is becoming fundamental in the scaling problem (1).

The works that will be presented are the ones that I looked at the beginning of my work to get an idea of what the field is studying at the moment. It is a very active area and many articles are published every month in the biggest journals such as the "*IEEE: Internet of Things Journal*" (31) and presented at the bigger conferences, such as USENIX's (32). It is therefore important to note that finding one groundbreaking solution that fits every case is highly unlikely, as has been observable. There is a general lack of comparison between the various recently published solutions since they make up different approaches to tackle the various problems, all of this while using different benchmarks (like some using DeathStarBench (33), TeaStore (22) or Sock-Shop (34) which is now deprecated).

2.5.3 State-of-the-Art Proposals for Scaling

Smart HPA: A Resource-Efficient Horizontal Pod Auto-scaler for Microservice Architectures (2024)

In "*Smart HPA: A Resource-Efficient Horizontal Pod Auto-scaler for Microservice Architectures*"(35), a threshold-based rule is proposed. The key starting idea is to take the standard Kubernetes HPA and enhance its capabilities in resource analysis and management.

- **What it addresses:** HPA based solutions, as stated beforehand, scale single microservices and by default can only consider natively **per-microservice** metrics, namely CPU and Memory utilization. Since it is using a reactive method, as also stated beforehand, this can lead to **over-scaling** or **under-scaling** in case of respectively high spikes or low spikes, whereas the general utilization might not be following that tendency at all, creating delays for the stabilization of the application and eventual problems in handling the load correctly. There is also the problem of it not being able to go over the application imposed limits if needed.

- **What is proposed**: A tool, named **Smart HPA**, which still uses an **heuristic** rule to scale and redistribute resources, but it does in an more efficient way than the standard Kubernetes solution. It also implements a **hierarchical structure**, so that the various individual autoscalers communicate with an higher entity that knows what every autoscaler has seen from the chosen microservice and aids in the scaling decision by giving its verdict based on the resources analysis.

- **What has it been tested on**: This tool has been tested on a Kubernetes Cluster on which the application *"Online Boutique"* (36), loaded with **Locust** and compared by just using the CPU Utilization **per microservice** as metric on which to scale, and to ensure reliability multiple tests have been conducted.

- **Their results:** On the tests, Smart HPA outperforms the standard one by **5x** in terms of resource over-utilization and **2x** in over-utilization time and in over-provisioning by **7x**, improving resource allocation by **1.8x**, eliminates is and extends over-provisioning time by **10x**. Their main focus is then on resource efficiency as stated before. They also remark how horizontal scaling time is a problem that needs to be addressed in future works. From this premise, some insights will be given in 3.5, based on my work.

- **Some remarks**: while it being a reliable better solution than the standard HPA, it does neither use AI nor time-series analysis, which is already slicing off a lot of possibilities on the side of workload prediction for scaling. It also still fundamentally considers scalers at microservice level, even if they took steps to make it better with the hierarchical structure. Furthermore, they are not considering SLAs, which are a secondary objective one method should pursue to guarantee better performances based on user needs.

μScaler: Automatic Scaling for Microservices with an Online Learning Approach (2019)

In "*μScaler Automatic Scaling for Microservices with an Online Learning Approach*"(37) , a scaler that continuously adapts to the new workloads patterns is introduced, optimizing resource consumption dynamically. It is a ML-based solution.

- **What it addresses:** It addresses the **limit** of **reactive** methods, and so mainly threshold-based ones, to not being able to predict the workload that the application might be facing, being unable to provision the right amount of resources in the right time so that the it can be absorbed correctly.

- **What is proposed**: It proposes an **Online Learning Algorithm** that takes first collects workloads and microservice metrics like latencies percentiles, then determines the service needing to scale thanks to the usage of an **Online Bayesian Function Optimizer** and finally acting by taking the **horizontal** scaling needed. The main goal they pursue conducting a **SLA-oriented autoscaling**, with the premise that by horizontally scaling microservices in an application, eventual request congestions can be dealt better with, leading to better a better respect of the boundaries of an acceptable level of the request latency in this case. In fact, this scaler does not rely on metrics like resource usage, but SLA boundaries.

- **What has it been tested on**: The method has been tested on the *Hipster-Shop* benchmark (38) on a Kubernetes-based cluster with Istio.

- **Their results**: They compare their work with other scalers, namely Amazon ones, but not with the classic HPA, even if they apply Horizontal Scaling. It worked better than the state of the art scalers of Amazon at the time. It is a good work that can inspire on SLA based autoscaling, specifically on latency.

- **Some remarks**: While it being a good work overall, it lacks comparison with more tools. It also still considers, like in SmartHPA, microservices to be scaled individually, even if it took steps to consider things more globally thanks to the usage of the definition of service power and of microservices that **actually** need to be scaled. On the positive side, it doesn't present any hard Deep Learning based solutions, which could be a plus on a level of applicability, at least in my opinion.

HANSEL: Adaptive horizontal scaling of microservices using Bi-LSTM (2021)

In "*HANSEL: Adaptive horizontal scaling of microservices using Bi-LSTM*"(16), a load prediction ML model is presented along with a scaling framework that leverages that information to

scale in an elastic and hybrid fashion microservices in an application. It is then a combined ML + RL algorithm.

- **What it addresses:** The lack of a solution that would **combine** a **reactive** (based on the resources consumption in the moment) with a **proactive** (based on prediction of workloads) approach motivated this work. Additionally, they state that the methods that came before them cannot reliably model what really happens in a data center, without using simulation data.

- **What is proposed**: A scaling method that combines both. An **offline model** training that focuses on the historical resources usage (like CPU and Memory) is **combined** with a **simulated online training model** that predicts load and other things like SLA monitoring. The key feature that should be noted is the usage of a **Bi-LSTM Network** to learn the time dependencies between states, leading to accurate load prediction thanks to the usage of the "attention mechanism" (39), a groundbreaking concept that was born in the field of Natural Language Processing but that is now expanding its reach to other fields like this one. For the act of advising an action after the analysis of this prediction, they also devise a Multi Agent Reinforcement Learning Problem to study the impact that this information should have on each microservice and then advise the hybrid scaler component that will take this decision and counterbalance it with the analysis made in the reactive part based on the study of instantaneous resource usage. Scaling is both horizontal and vertical.

- **What has it been tested on**: They use the *"Alibaba Cluster Trace"* (30) (cited before in 2.4.3) to generate data for the offline reactive part of the training and they use the *"Sock Shop"* (34) benchmark to generate the future load prediction set used for the proactive part. They also tested the same environment scenarios with other predictive time-series analysis methods like ARIMA and Linear Regression Models, and finally also with other (at the time) State of the Art works like ATOM (40) or MLScale (41), which are not presented here.

- **Their results**: The results have been promising: in terms of CPU utilization and SLA conflicts, over time HANSEL wins greatly over the others, converging faster to an optimal value of CPU usage of the microservices (around 70%) in less time than the others and an overall low value of SLA conflicts (a few in the spans of 5 minutes compared to other methods which can reach higher values from the start), which are the violation of said SLAs multiplied by the average time that they last.

- **Some remarks**: It is definitely an harder approach than the other ones, but as demonstrated it works well in a lot of scenarios. It is moreover a method that combines instant knowledge (CPU, Mem Usage...) with SLAs violation constraints put from the user and predictive load. It is per se a good work, but it being an application of Machine Learning Models like **Bi-LSTM** and **Attention**, it needs to be adapted to many situations and it obviously needs time and generality of workload environments to simulate. But the intuition of "learning" the workload seems promising. We will see another application of this more in detail in (3) later in another proposal. As for other works, this paper also presents a per-microservice individual scaling.

CoSCAL: Multifaceted Scaling of Microservices With Reinforcement Learning (2022)

In "*CoSCAL: Multifaceted Scaling of Microservices With Reinforcement Learning*"(42), a prediction algorithm is presented (so a **proactive** solution) and uses this and other data to train a Reinforcement Learning algorithm to enact the decision. It is then a combined ML + RL solution.

- **What it addresses**: The work tries to address the right guarantee of a good Quality of Service without over-provisioning, like the other papers. It is then a **SLA-centric** work.
- **What is proposed**: It proposes a **two phases**-approach: a time series analyzer that analyzes the processed traces from the microservices and generates data predictions about the next states. This and other data about the performance are given to a unit that processes this and decides on how much to scale. This time, though, scaling different: Horizontal Scaling is referred to engaging more physical nodes that can contribute to the whole application run, while Vertical Scaling is referred to giving more resources to the microservices running on those nodes (i.e. assigning them more resources out of those available in the respective nodes). There is finally a third scaling option that is also learnt by the Reinforcement Learning algorithm which is **brownout**: this is more similar to the Horizontal Scaling we have seen in this context. Replicas of the same microservices are created, but this time they are created before-hand and deactivated and activated on command, so they take less resources on the nodes and in this way time-related issue to containers initialization is partly solved.
- **What has it been tested on**: They tested this on the "*Sock-Shop*" (34) application and then using various tools to monitor the metrics on the cluster. Load has been generated with Locust.
- **Their results**: They compared their approach to other native scalers like ones in Docker Swarm or Kubernetes, or HyScale (43) (which is another solution based on hierarchical threshold-based scaling not shown here). The results have been given in terms of **failed requests**. With their approach on a range of experiments, the number of failures gets reduced by an order of magnitude (from 10^5 to 10^4 in the span of some hours of experimentation. The **response times** (which represents **end-to-end latency**) have also become better compared to the other ones.
- **Some remarks**: The comparison looks promising but still lacks generality. Moreover, as for other works, this paper also presents a per-microservice individual scaling. Finally, while it works and it performs well on its own, the reward function of the reinforcement learning model is rather simplistic, not accounting for possible behaviors that a trained scaler would have (this will be seen in Chapter 4).

FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices (2020)

The two following papers have more been the object of my study and the ones I have to thank the most for ideas during the whole project. They have been published by the same main author.

In "*FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices* "(2), a **combined ML + RL** solution is proposed, leveraging various concepts in the field like *Critical Paths* and *Anomalies Injections* to create a framework that can learn from the problems in the application and enact the correct scaling decision thanks to the usage of Reinforcement Learning Agents. Their solution tries to counter **SLA violations**.

- **What it addresses**: As previous works, in FIRM the authors also focus on SLA, specifically on the violation of those. One example of this, as said before, can be the **response time** (or end-to-end latency), which is a problem to the end user. Their work is, as the others, motivated by the **inefficiency** of current **heuristical threshold-based rule**s to effectively scale microservices.
- **What is proposed**: A two phase ML + RL framework is proposed:
  - The machine learning part learns and recognises special execution history graphs called **Critical Paths**, (which are the critical microservices through which a high-latency request passes on its execution), that are later passed through another algorithm that, thanks to the usage of indicators like congestions intensity and relative importance in the contribution of latency to the whole Critical Path, extracts the **Critical Microservice** from it thanks to the usage of a pre-trained **Support Vector Machine**, which is a machine learning classifier. This information is subsequently passed to the **reinforcement learning Agent**
  - The **Agent** takes the information about the Critical Microservice and the data that has been collected from the application environment (namely instantaneous information like CPU, Memory, Disk and Network Usage) and converts that into a scaling action. Since the reinforcement learning Environment works through the definition of states and rewards, at each timestep the used states are mainly Resources Utilization (RU_t per microservice, same definition as in Equation 2.3, technically $\in [0,1]$) and SLA Preservance Ratio (SP_t, taking the definition given in 2.4.1). The goal of the agent, for the microservice, is to maximize the reward function :

    $$r_t = \alpha * SP_t * |R| + (1-\alpha) * \sum_{i}^{|R|} \frac{RU_i}{RLT_i} \tag{2.5}$$

    where RLT are the various Resource Limits associated to the Resource Utilized. The full definitions of these concepts will be presented in 4, but it is a function that explains how well the scaler has done based on those states. In this case the objective is to have a high "reward" which translates in having a high resource utilization over the pods and a high SLA Preservance ratio. The action that the agent takes can be vertical and eventually horizontal if vertical is not enough anymore and if needed.

To further guide the learning process of the Support Vector Machine, they also implemented an **Anomaly Injector**, which targets specific microservices and makes their execution faulty/worse, giving some ground-truth to the model so that it knows which is the actual Critical Path that is detected.

- **What has it been tested on**: The framework has been extensively tested on various microservices benchmarks: namely the active "*DeathStarBench*" benchmarks (33), which are "*Social Network*", "*Media Service*" and "*Hotel Reservation*", and on "*Train Ticket*"(44). For the training of the Reinforcement Learning Agents, an extensive study on their convergence and the usage of transfer learning (a particular kind of reinforcement learning training) is done using a benchmark to train (*Train Ticket*) and another to test (*DeathStarBench*). They use Prometheus to gather metrics on a regular basis (seen in 2.3.3).

- **Their results**: They compared their method with Kubernetes **Autoscalers** and an **AIMD** (Additive-Increase, Multiplicative Decrease) scaling method (not really specified of which kind), and found that it outperformed both baselines leading to respectively to **16x** and **9x** less SLA violations, while also increasing CPU utilization up to **33%** and other various results.

- **Some remarks**: FIRM proposes an extensive framework that tries to address the problem proposing valid ideas that can be extrapolated and enhanced in a modular fashion. Still, the problem with such a framework (as I have found out) is the applicability: every part needs to be deployed single-handheldly and unfortunately the artifact that they left does not work anymore, since they migrated to new projects. Moreover again like others, microservices are still considered individually, even if efforts are made to filter those who are not actually in need to be scaled. Another limitation is posed by the fact that it taking time, scaling could theoretically not be needed anymore after the scaling has been done, because the SLA violation might already be over by then it potentially being a spike. The problem with this is that, even using ML and RL, at its core this work is still of a "reactive" type. No proactive load prediction is made, so no predictive scaling can be done to counter eventual SLA violations encountered on the run.

AWARE: Automate Workload Autoscaling with Reinforcement Learning in Production Cloud Systems (2023)

This final paper presented here is the one on which I based the second solution proposed in this work. Their reinforcement learning code has been vastly adapted by this work, which is open-source on their GitLab (45). More specifically, I thank them for providing a working version of the agent implementation code of the PPO model, that I used to study on and later re-wrote for my reinforcement learning solution, explained in 4.2.3.

In "*AWARE: Automate Workload Autoscaling with Reinforcement Learning in Production Cloud Systems*"(3), yet another **combined ML + RL** solution is proposed, leveraging some concepts presented in previous papers to create a framework that can learn from the problems in the application and enact the correct scaling decision. Their solution tries to **counter SLA violations**, optimize resource usage and solve some of FIRM's issues like the non predictability of the load. It is in fact made by the same team, so this paper is a natural sequel to the last one, even if with a complete change of solution.

- **What it addresses**: Like in FIRM, this paper addresses **SLA**, particularly focusing on minimizing their violation. In doing so, they also try to work on the instantaneous usage of resources, this time focusing more on the most important ones, they being **CPU** and

Memory utilizations. They also try to address another big problem of reinforcement learning. This topic is pretty extensive and is hard to apply in production since it's **training** and **workload dependant**. This means that every time that context is changed, the optimal solution is not guaranteed to work in the changed environment, because it is hard and expensive to train models that could work that generically, with also being easy to deploy and to adapt. They finally also address FIRM biggest problem, which was its lack of workload prediction factor that could aid the Reinforcement Learning Agent on taking a **proactive** decision and scale in **hindsight** of eventual **SLA violations**. This complete change of direction is seen in the first phase of the following solution.

- **What is proposed**: In AWARE, a new ML + RL framework is proposed.
  - The machine learning part is composed by a **metalearner** that helps the reinforcement learning in gaining more context about the possible combinations of *(application, workload)* pairs by generating an embedding that is then fed to the **reinforcement learning agent**. In short, when an **application** goes through a certain **workload** over time, it can have various characteristics that might be similar to another **application** with another **workload** that would require then the same treatment, even if never seen before. The metalearner here is trained with a big dataset of possible combination of these two composite variables and learns how to create an embedding that is a translation to a big dimensional vector of that aforementioned couple. Similar application with similar workloads will be projected onto a space in which they will reside in a close area, while other may be still be close or away. The embedding then says to the RL agent how close to another already-seen state the application is. This overcomes FIRM's retraining problem: with this, retraining times and costs are less, while also they being more probable to converge to a generally better optimum.
  - The reinforcement learning solution is composed by a *Proximal Policy Optimization* agent (46)(**PPO**) that receives as input an embedding from the metalearner and informations about the status of the application thanks to a metric retriever system. Then it specifically waits some time to get those metrics and makes its decision based on those. The agent then makes its decision based on its current understanding and after a while learns based on the past experiences. The implemented model is one of the most recent on which research is being made also by big companies like **OpenAI**. The environment provides a reward function, similarly as in FIRM, that tries to push the agent to scale based on high resource utilization and high SLA preservance ratio. The states and the associated reward formula are the same as in FIRM in fact, as for the last one written in 2.5.

    The agent training time is eventually even more reduced by another component: the **Bootstrapper**. This internal component checks if the agent is "deranging" from the norm, which means that it is taking too many random decision that don't effectively bring a high performance. This always happens at the beginning, where the agent is usually not trained. This component then just activates a new custom Kubernetes Autoscaler, called *Multidimensional Pod Autoscaler* (**MPA**), that will scale instead of the reinforcement learning agent. This definitely gives a boost to the training time, giving less focus on the "exploration" phase at the beginning for a little more

guidance in the training.

- The aforementioned MPA is an "in development" solution to try and mix the benefits of having a VPA with an HPA. It was developed originally by Google for their Google Kubernetes Engine (**GKE**)

- **What has it been tested on**: For metalearning, a big training dataset using **16** different applications segments arranged in different combinations to form full applications is created. For the metrics, they are taken from a Prometheus. The applications that it has been tested on have not been thoroughly specified in the paper.

- **Their results**: Their main comparison was with their work made in FIRM. In this case the main result is the time it takes the whole model to converge to an optimal status right away. It is shown that it outperforms FIRM adaptation speed by roughly **5x**, while also reducing **SLA violations** by **7.1x** during that time. Other tests have also been conducted and they all show the superiority of the model towards FIRM.

- **Some remarks**: The work seems well structured and the solution innovative, but it left some open points: it could have been tested on more "standard" benchmarks like its predecessors and give some more comparisons with other reinforcement learning-based solutions, even though I acknowledge that comparing solutions in this specific topic can be a bit problematic and lengthy. Finally,It still consider microservices scaling individually, like all the other solutions. In this case specifically, there is one scaling reinforcement learning agent that is deployed **per microservice**, making it necessary to train many agents at once.

  t takes some time and knowledge to get through it with the code, but it is the only paper that left a good enough artifact on GitLab with many different things to try out and to learn from.

Chapter 3

1st Problem: Grouped Scaling

3.1 Problem presentation

As presented in the last chapter, scaling is a faceted problem, with many problems to possibly solve such as efficiency of it in terms of performance times, effective load absorption and many more.

One of the aspect that I was tasked in exploring was the possibility of applying a scaling in a grouped/coordinated way. This means taking the **orchestrator API** and issuing commands to it to **scale more microservices** in one go by just making **one single API call**, by also eventually exploiting eventual dependencies information. This first problem will be presented in this chapter. This mechanism, that from now on will be called **Grouped Scaling**, has proven to be a feasible task.

Various goals have been defined throughout the whole development procedure:

1. **How** and on which basis we **define** a **group** of microservice;
2. Which **metrics** and how to retrieve them;
3. Define a way to **scrape** the metrics and **regularly retrieve** them;
4. Define a **controller entity** that process this data for groups;
5. Define the policy to decide on the scaling action;
6. Define a way to apply those actions in Kubernetes in a **safe** manner.
7. Devise a way to simulate a typical workload in the environment.

The points covered in this chapter are 1,6. The problem will be presented with an explanation on what a group of microservices is in 3.1.1, why we could need this mechanism in 3.1.2 and finally how the mechanism is implemented in 3.2. No actual metrics or policy is used in this case to scale: the main objective is proving that this is achievable in practice. Since the other points are not covered here, experiments in this chapter will mainly focus on showing that grouped scaling works and what insights we can get about scaling times and its theoretical efficiency. These points will be covered in 4.

3.1.1 Definition of a Group of Microservices

In the Kubernetes, a *"native"* grouping mechanism for Deployments does not exist. The only thing that could mostly get close to it is labelling the Deployments in the *.YAML* files with labels that can be used to filter various microservices as if they were in groups. On a higher level, grouping has to be made and implemented from ground-zero.

Thus, the definition of "Group of Microservices" has to be made on a higher-level.

We could define it as a group for which microservices inside of it may have a special relation to one another (i.e. dependency) that is deemed by human decision or an algorithm (i.e. a ML algorithm) that it should be considered together.

$$G_x = \{ms_1, ms_2, ..., ms_n\} \tag{3.1}$$

where n is the dimension of the group.

Resource metrics for a group

Since a group of microservice is defined, the metrics to be used to represent the state of a group also need to be defined. In this part of the work, these metrics will not be used, but they will be used in the solution proposed in Chapter 4.

In 2.3.3, we went through some common metrics used in the field of scaling. As stated, the **relative resource utilization** presented in Equation 2.3 is a useful metric to base the scaling action on. An interesting and easy metric to define for a group would be the **resource utilization per group**.

$$RRU_{group} = avg_{group}(RRU_{microservices}), \forall microservices \in group \tag{3.2}$$

This formula doesn't account for the **eventual weight** that various microservices might have in the group. This is due to the assumption that microservices in a group should have similar characteristics so that they are put together. A **future work** could study this direction as well as a way of defining groups.

3.1.2 Why grouping microservices?

In 2.4.2, we quickly explored how microservices are still intertwined between each other. With tracing and by creating a service dependency graph and eventual execution history graph for various requests, we can can see this in action.

As an example, two service dependency graphs from the benchmarks of TeaStore (22) and Social Network from DeathStarBench (33) are shown in Figure 3.1.

An example on TeaStore

Let's take TeaStore as an example. As visible in Figure 3.1, the *WebUI* microservice calls almost every other microservice, while there are other microservices which are not calling any other ones, like Registry. A natural question now arises: how can it be possible to actually determine the **entity** of these dependencies?

One possible answer is through *Trace Analysis*, introduced in 2.4.3. This solution is not covered by the scope of this report, so it is not possible to go down this path. Another possible

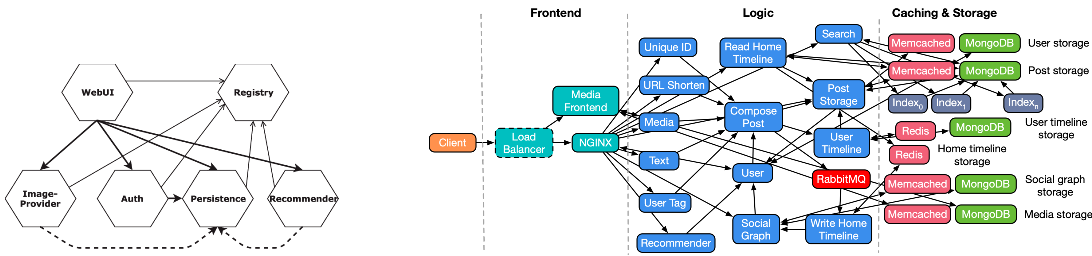


Figure 3.1: Dependency Graph for TeaStore and DeathStarBench's Social Network

answer, which is easier to visualise, is to track the instantaneous performance counters of the application and make deductions based on those. To show this, a quick load injection was performed on the TeaStore application using Locust (28). The application presented no limits on the resources. The application was injected for a time of $120s$ going from 0 to 100 simulated users making a full round of requests during that time. CPU and Memory Usages have been measured and no limits have been put in the TeaStore Deployment file.

The results are taken thanks to Prometheus(7) and are shown in Figure 3.2. The metrics are absolute in this case, so the unit of measurement for the CPU is **CPU cores per second used**, CPS for short, while for Memory is the GBytes of **Allocated Memory**. They are shown over a period of time of 200 seconds. As it is possible to see, the CPU and memory usage for **teastore-webui** respectively peak at values around $4CPS$ and $2Gbytes$ at $100s$, whereas **teastore-auth,persistence,image** all peak around $1.5CPS$ and $1.5Gbytes$ at the same time. As noted, the various microservices share a similar behaviour but different magnitudes of resource utilization, especially for CPU utilization. For CPU we can see that the total usage

In here we see the advantage of microservices, as splitting the various function of the application is making the different resource utilizations evident. We also see another behaviour: Some microservices have similar resource utilization as others in shape and magnitude.

This brings a plausible justification as to why are we considering grouped scaling: even if we don't address the problem of dependencies through a formal study, it can be insightful studying its technical feasibility and features just by looking at the performance metrics alone. For example, at a given instant, the HPA object related to the **teastore-image** microservice would **most probably** give the same answer as the HPA object related to the **teastore-auth** or **teastore-persistence** microservices. Why deploy more scalers for all the individual microservices when we can deploy some **custom scalers** that do the same action for those microservices of which the performance counters behave in the same way for a given load? To this we answer with the Grouped Scaling implementation.

3.2 Proposal

The proposed solution for answering the **Grouped Scaling** technical feasibility question is the creation of a new scaler object from zero, deployed as a **Kubernetes Deployment**. Because of this:

- No algorithm is proposed here to dynamically form the groups; **static group** divisions will

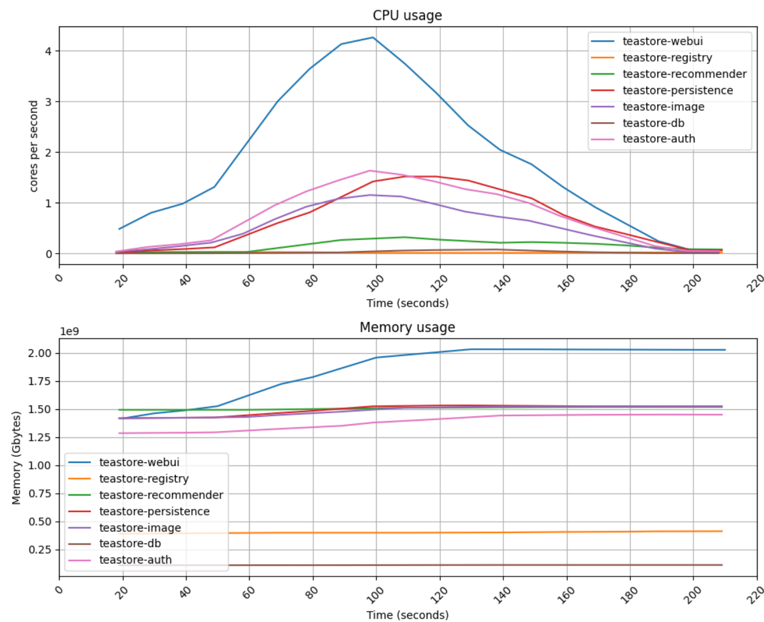


Figure 3.2: Resource Usage from the TeaStore Application under a light stress

be used, based on the considerations done in 3.1.2;

- No actual policy is decided for the scaling. The scaling operation has been first applied in a random way to see if it worked, and then in a more cadenced manner for experimentation. Policy is not needed as we just needed to see if it worked and such a discussion would have been made anyways in Chapter 4.3. More in 3.5.

The type of scaling that has been used is the **Horizontal** one, as it is the most used in production and studied in research as seen 2.5.

To first implement it then, the concept of Control Loop Pattern and thus Kubernetes Operator are needed.

3.2.1 The Control Loop Pattern and the Operator

In the world of Electronics or I.T., it is not uncommon to encounter the concept of the *Control Loop* pattern. It is needed to continuously (or at intervals) check the state of the environment to monitor and issue actions based on the value of the registered state.

If brought down to Kubernetes, this translates to an **Object** that tracks at least one resource types and that has a **desired state** entry to which it tries to take the controlled resource to. To do so, API calls are issued, either to the resource or to the Kubernetes cluster.

The **Operator** is the software implementation of the controller, made in Kubernetes. They are defined as "clients of the Kubernetes API that control a Custom Resource" (47).

3.2.2 Grouped Scaler

For the part of this proposal then, the Grouped Scaler Operator will be presented, from its technicalities to how it has been used to gain insights on how grouped scaling is affected by the cluster conditions.

How does it work

A quick explanation will be given.

1: Initialize and connect to the Kubernetes cluster.
2: Read initialized groups and check the application status.
3: **if** the application is started **then**
4: Start threads equal to the number of groups.
5: **for** each thread **do**
6: Manage one group.
7: **while** True **do**
8: Apply a policy (e.g., random).
9: **for** each group i **do**
10: Scale the group using M API calls where $M = |G_i|$.
11: Each thread waits until all microservices in the group are scaled.
12: Threads synchronize and wait for each other.

The following section will talk about implementing this solution in a Kubernetes environment.

3.3 Operator Solution Implementation

To construct an Operator, the first thing that has to be decided is the programming language to use. **Kubernetes** is fully written in **Go**, so the preferred one should be that, but it is not a language that is extensively taught in standard education, not as much as C++ or Python. Moreover, programming an Operator from zero is quite hard if we need to account for all the events that happen on it that need to be listened for by our custom scaler. Fortunately, the Kubernetes Community has devised some frameworks to solve both problems. There are now a varieties of frameworks to program an Operator from scratch in various languages. A list of them is available on the Operator Pattern webpage (47).

Among all languages, **Python** was chosen: I subsequently chose *KOPF*: *Kubernetes Operator Framework*. It is a good compromise if the objective is to build a prototype to see if something could potentially work in an Operator fashion, without going through all the technicalities of the Go Language.

The artifact of the code is available at the ERODS's team personal GitLab space.

3.3.1 KOPF

KOPF usage is straightforward: it can be easily installed as a Python Package and easily implementable in any Python Script as a library. From there, one just has to follow the guidelines

provided on the official Documentation(5).

The framework provides with an internal engine that manages the loop process. All that needs to be defined by the programmer is the events that the loop will need to listen for. An **Event** is a Kubernetes Object that logs something happening in the cluster, namely a change of state. The main events that we mainly see are of three types, and are the ones that comprehend the Objects' lifecycle:

- **Creation**
- (Notice of) **Deletion**
- **Update**

With this in mind, what the framework does is providing a series of background-active functions templates that:

- Natively implement the **loop** functionality;
- Can be extended, creating new fully fledged functions that will be called **when** a specific type of Event is registered in the Kubernetes Cluster;
- Logs the internal and Kubernetes errors, debug and eventual additional logging.

The second feature is the one exploited by the developer. It is the core definition of the custom functionality. To extend a function, KOPF defines Python decorators that can be put at the beginning of each function declaration. Here we give a simple example:

```
1 @kopf.on.startup()
2 def startup_fn(memo: kopf.Memo,**kwargs):
3     print("Starting up...")
```

In this case, this means that the function **startup_ fn** will be called at the beginning of the Operator lifecycle. In this case, the decorator has no arguments. Let's now see another example:

```
1 @kopf.on.delete('apps', 'v1', 'deployments')
2 def app_delete_handler(spec, **kwargs):
3     print("Deleting deployment")
```

In this case, the function will be called just when the Operator detects that an Event of type **delete** is applied on an Object of type **deployment** in the API group of **apps/v1**. In the case of an application composed of N microservices, all deployed as **deployment** Objects, the **app_ delete_ handler** would be called N times.

3.3.2 Phases of Running an Operator through KOPF

KOPF based Operators are run through the usage of its specific command. An example of such simple command is the following:

```
kopf run operator.py
```

At a debug level, this command is easy and efficient, as it is designed to be run in the Kubernetes cluster's control node, on which the script can have full visibility of the cluster state and deployments status.

For release, instead, things are a bit different: the usual procedure involves encapsulating all the script with all dependencies installed in a Docker Image, which is then uploaded on a Image Registry for it being then pulled when needed from the Kubernetes Cluster through the usage of the application .yaml file.

Additionally to the deployment Objects that are created for the microservices, in this case, additional entities are needed. These are called **RBAC Roles**. RBAC stands for *"Role-Based Access Control"* and is a way to regulate access to Kubernetes resources from other entities or resources in the cluster (48). Since the Operator needs to read events from the cluster and send API calls for certain actions to the resources, it needs the RBAC Roles for the right actions on said resources. Those are also provided in the artifact I left.

Once the Operator is run, it goes through three main phases of its own, namely:

- **Startup**
- **Control Loop cycle(s)**
- **Shutdown**

Startup: Initial configuration and group setup

The startup procedure is run at the beginning, in the following function:

```python
@kopf.on.startup()
def startup_fn(memo: kopf.Memo,**kwargs):
    print("Starting up...")
    #configuration object startup
    #configuration kubernetes client
    #read groups from txt
```

I created a configuration object in which I store all the configurations needed, like the Kubernetes Python client configuration (needed to connect with the cluster and send the API calls) which is created right after by reading the **kubeconfig** file of the Kubernetes cluster (which, as the name says, holds all the internal configurations of the cluster). Finally. I read the statically defined groups from a .txt file.

Control Loop Cycle Implementation: Daemons

The main entity that we have to look for to implement the control loop pattern in *KOPF* are **Daemons**. They are entities created with the same decorator seen before that are also passed, as argument, a bool variable that in the background is updated when the loop has to be stopped. This variable is managed by the *KOPF* background engine, so its value setting is not of our concern. We just know that it switches whenever the daemon has to be stopped.

For our needs, a **daemon** will be the actual scaler entity, i.e. the aforementioned thread. Since we want to issue a scaling call for all the microservices in a group, it is obvious that we need to create as many daemons as the number of groups. Here is a resumed snippet of the **daemon**.

```
1   @kopf.daemon('apps', 'v1', 'deployments')
2   def daemon_sync_scaling(memo: kopf.Memo ,stopped, **kwargs):
3       config_operator = memo["config_operator"]
4       app_name , run = #get it from the kwargs
5       label_selector = f"app={config_operator.app_name}"
6       if app_name == config_operator.app_name:
7           config_operator.add_run(run)
8           selected_group = None
9           for index, inner_array in enumerate(config_operator.groups):
10              if run in inner_array:
11                  selected_group = index
12                  break
13          if selected_group is not None:
14              if get_thread_dict(selected_group) is None:
15                  modify_thread_dict(selected_group, run)
16
17          if selected_group is not None and get_thread_dict(selected_group) == run:
18              while not stopped:
19                  stopped.wait(ACTION_INTERVAL)
20                  scaling_group(selected_group, config_operator, memo)
21
22  def scaling_group(group, config_operator, memo):
23      num_replicas = #CHOSEN WITH SOME POLICY
24      try:
25          running_for_group = 0
26          for microservice in config_operator.groups[group]:
27              if microservice in config_operator.run_states.keys():
28                  try:
29                      running_for_group += 1
30                      resp = config_operator.appsV1.patch_namespaced_deployment_scale(
31                          name=microservice, namespace=config_operator.namespace, body={"spec": {"replicas": num_replicas}}
32                      )
33                  except Exception as e:
34                      print(f"Problem in scaling: {e}")
35          groups_copy = config_operator.groups[group][:]
36          while len(groups_copy) > 0 and running_for_group > 0:
37                  time.sleep(TIME_PRECISION)
38                  for microservice in groups_copy:
39                      if microservice in config_operator.run_states.keys():
40                          try:
41                              resp = config_operator.appsV1
42                                  .read_namespaced_deployment_status(name=microservice, namespace=config_operator.namespace)
43                              ready_replicas = resp.status.ready_replicas
44                              if ready_replicas == num_replicas:
45                                  groups_copy.remove(microservice)
46                          except Exception as e:
47                              print(f"Problem in reading replicas: {e}")
48      except Exception as e:
49          print(f"Problem in scaling: {e}")
```

In this context, a daemon is created for every microservice. A filtering mechanism is then run. Just one daemon per group will survive, while the others will get killed. For that group, then, the **loop** will consist in a **waiting** action (usually more than 30 seconds, to wait for the application to stabilize) and the actual scaling action.

The scaling function is quite simple. In this case: For the group the number of replicas to scale to is chosen. For all the active microservices in the group for which the daemon has the duty to scale, it sequentially sends an API call to the Kubernetes API. The second loop in the function just loops continuously until all the replicas in the group are ready. As stated in 2.3.2, horizontal scaling takes time. For us, waiting for the replicas in the group to be ready indicates that the scaling action to be completed.

The **Shutdown** part will not be presented, as it does not present any valuable logic worth to be included.

3.3.3 Future for the Grouped Scaler

The Scaler Diagram project is displayed in 3.3:

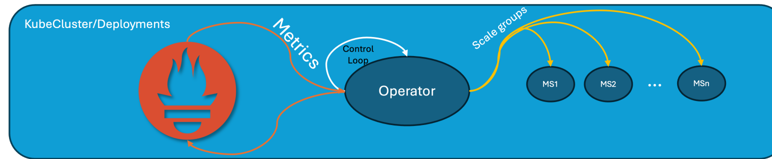


Figure 3.3: Grouped Scaler Operator schema

The original plan for this product was to make it more similar to the whole standard Kubernetes scaler like **HPA** and **VPA**. In those, as stated in 2.3.2, we need live instantaneous metrics for these two tools to compute the number of replicas to scale to and the new resource limits. How are the metrics provided? The HPA and the VPA require a service called *Metrics Server* (25). This service exposes metrics that it gathered by asking the cluster nodes for them. Specifically in each node, the various deployments will eventually have to expose their metrics through their container runtime and therefore their Pods. It will be another internal service's duty to scrape those metrics and expose them to the outside environment through the usage of the right APIs.

For our operator, the original plan was not to use Metrics Server, but to leverage another Metrics Scraper and Provider: Prometheus(7).

In the end, its usage was not implemented in the code since no real policy that would have used metrics has been implemented.

Future works could include an implementation of some kind heuristic-based rule, Prometheus integration and dependency-based scaling study.

3.4 Experiments

The test has been conducted on the TeaStore Application Benchmark, presented in 2.2.4.

3.4.1 Goals of Experiment

The main experiments goal, as stated in at the beginning of this Chapter, is to test if this worked. To test the correct functionality, running the script was sufficient. It effectively worked as expected, being able to sequentially send API calls to the cluster and initiate the due scaling procedures. The task has been proven to be technically feasible, proved by the corresponding code presented in 3.3.2 for the Daemon.

This experiments were also an opportunity to further study horizontal scaling and trying to characterise it a bit more. To do so, this Operator was deployed in a different version that would record the times of scaling. The objective has then been to get those times and analyze those results.

3.4.2 Analyzed result: Scaling Time

Specifically, the time that was analyzed was the **total scaling time for the group**, expressed as in Equation 3.3:

$$Total_t_{group} = \frac{min(t_ScalingRequest_{group})}{max(t_ReadyPod_{group})} \tag{3.3}$$

The creation of a pod goes through various states that are monitored by the Kubernetes Cluster. We define scaling as completed when a microservice associated replicas are all in the *Running* state. We therefore define the group to be ready when the last of its microservices goes into the same state.

To extensively test this mechanism, a reliable Kubernetes cluster along with powerful distributed nodes were needed. For this then, the French national testbed cluster for research was used, Grid'5000 (13), and as such it will be presented now.

3.4.3 Grid'5000

Figure 3.4: Grid5000

"Grid'5000 is a large-scale and flexible testbed for experiment-driven research in all areas of computer science, with a focus on parallel and distributed computing including Cloud, HPC and Big Data and AI" (13).

The cluster has a variety of nodes scattered across the whole French territory. To work on a node, one just needs to book it and a certain number of its host and deploy the experiments.

In this case, all the setup included getting an image to deploy on the nodes, setup all the packages and modules to run the Kubernetes Cluster and finally run the experiments. Everything I have ever produced and tested during this time has been tested on GRID'5000, as well as the work produced in 4.3.

The nodes in the cluster have a wide range of specs, based on the type of job that they are supposed to carry out:

- **CPU**: From $1core/CPU$ to $64cores/CPU$ for more specialized jobs. The ones used in these experiments can go from $8cores/CPU$ (Intel Xeon E5-2620 v4) to $32cores/CPU$ (Intel Xeon Platinum 8358).
- **Memory**: From 2GiB to 6TiB. The ones used in the experiments range from 64 to 512 GiB.

- **Other resources**: Some nodes are equipped with GPUs, but they are not of interest in our study, as our microservices benchmark only require CPU and memory. Disk is not important either, since we are not running benchmarks that use it intensively.

3.4.4 Experiments Setup

The experiments have been carried out on a total of 4 nodes of the chifflot Grid'5000 cluster in Lille, so using $6*(2CPUs$ with $12cores/CPU$ **and** $192GiB$ of memory). The experiments have been carried out once with the setup of **1 control node + 2 worker nodes** and another one with **1 control node + 3 worker nodes**.

Just 3 microservices were chosen and put in different group configurations, making up to 4 different experiment setups:

- **G1**: teastore-webui
- **G1**: teastore-webui, teastore-persistence
- **G1**: teastore-webui, teastore-persistence, teastore-recommender
- **G1**: teastore-webui
  G2: teastore-persistence
  G3: teastore-recommender

Rounds are defined. The groups are progressively scaled up to those values in the i-th round. The general round setup is $(1;2;5;10;20)$ for the first two experiments and $(1;2;5;10)$ for the last two, to not overload the Kubernetes cluster with replica creation.

For each group at iteration i, the correspondent daemon:

1. sets the number of replicas to 1 and wait for every microservice to be at that state;
2. take the time before the scaling and then scales to the number of replicas defined in the i-th place;
3. wait for all the microservices replicas inside of it to be of status **Ready**, finally taking the time of the last one which went to that state.

For it to be a reliable test, the experiments have been run a total of 20 times for each experiment setup, and then the times were averaged.

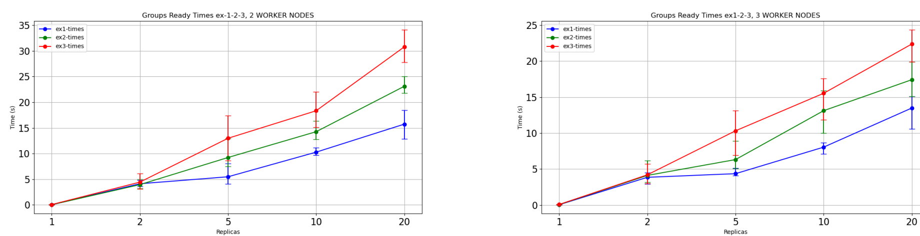


Figure 3.5: First 3 experiments times with 2 and 3 worker nodes

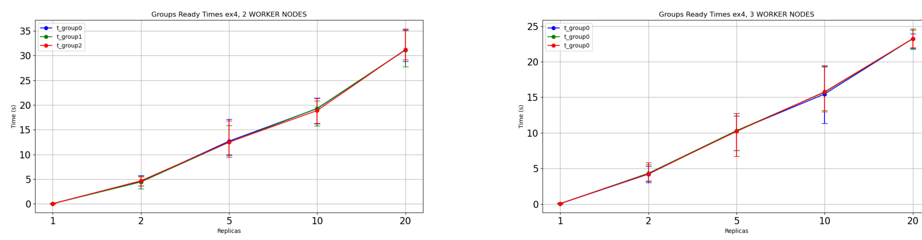


Figure 3.6: 4th experiment times for the various groups with 2 and 3 worker nodes

In the graph, on the X-axis we show the number of replicas of the corresponding round, while on the Y-axis the $Total_t_{group}$ for a specific group. The first three experiments have been grouped in one graph as each experiment was composed on 1 single group, therefore outputting one single averaged per-group value for the time. In those 3 experiments, it is visible how scaling time follows a linear tendency, as also how scaling times rise with the numbers of total microservices scaled at the same time.

In the case of 2 worker nodes in 3.5, scaling one microservice up to 20 replicas can take up to 15 seconds, whereas scaling three altogether up to the same amount takes up to 30 seconds.This last time is pretty significant. In the case of 3 worker nodes, for 3 microservices scaled up in a group, time is better than the last one: roughly 22 seconds compared to 30.

Experiment N4, shown in 3.6, shows that the same microservices in different groups all take the same amount of time to scale shown by the fact that the three lines for the three different groups overlap, and that the times results are equal to the ones of Experiment 3 considering both amount of worker nodes. The last fact suggests that scaling time depends on the total amount of microservices that are being scaled at a given time.

These results bear some insights.

3.5 Results and Insights

From the three results, three insights can be derived:

- **Insight 1:** Scaling times are **inversely** dependant on the number of worker nodes and therefore on how the scheduler handles them. More specifically, it is evident how a bigger number of worker nodes greatly affects the scaling times for more replicas rather than the small ones. This shows the internal Kubernetes scheduler's ability to distribute at its best possible capacity the various Containers and therefore Pods among all the worker nodes.
- **Insight 2:** Scaling times are **linearly** dependant on the **total** number of microservices to be scaled. As visible from comparing the **3rd** and the **4th** experiment, spreading 3 microservices in 1 group or creating 3 groups with 1 microservice each, even if being a edge case, shows how the abstract concept of grouping microservices doesn't substantially change anything at the level of the Kubernetes scheduler for this action. The cluster does in fact just see API calls arriving at it. The only difference is that for a group those are made sequentially, while for more groups those are made in parallel. But the evidence suggest that this does not affect the total behaviour of the procedure as these calls are not distant from each other. The **key factor** here is that the Kubernetes scheduler tries to **distribute** the pods creation and their "readyness" availability among the whole cluster. The total scaling times are then based on how much "**total scaling**" is currently being applied in the cluster.
- **Insight 3:** Horizontal scaling can take up a significant amount of time for certain types of applications. If the application is going under some serious stress and the application is throttling, waiting such a long time can cause a long disservice that is translated in significant SLAs violations. This is the proof of the need and effectiveness of the aforementioned "proactive" scaling, so much studied in (16) and (3), to make two examples.

Acknowledged these results, this dissertation now moves in another direction: trying to give a policy to grouped scaling, as it is for now missing. The problem was addressed through Reinforcement Learning, as many State-of-the-Art solutions did recently.

Chapter 4

2nd Problem: RL-based Coordinated scaling

4.1 Problem presentation

As seen in the last chapter, grouped scaling is technically feasible, and we even gathered some results about horizontal scaling times and produced some insights from them. But the **Operator** lacks a policy on which to act. Generally, a **policy** is defined as the logic following which we get the result of the action to take on the environment.

In the case of the HPA this has been defined as the formula seen in 2.3.2 for **horizontal scaling**. Inspired by (16),(2) and (3), the next step of this dissertation is to show how a possible implementation of a **Reinforcement Learning Framework** could be carried out, moved by the knowledge that this type of solutions are getting quite the fame in the field. This kind of solution is proven to be a viable option to create a **policy** for the scaling action, so it has been explored in the context of grouped scaling after proving it was possible with the operator.

This chapter covers all the questions left open in 3.1.

This chapter will quickly present all the concepts related to the field, subsequently presenting a solution on how to implement such a framework. In order, a **reinforcement learning** introduction will be made, with all the necessary knowledge about it in 4.2. Then a quick proposal will be made in 4.3 and the implementation ideas and schema are presented in 4.4. Finally, a round of tests and experiments are presented in 4.5. The goals of the experiments have been multiple:

1. **Main**: Test if the concept and the idea of grouped scaling could be applied with the aid of reinforcement learning. So again, like for the operator, a proof of concept, trying to put together all the knowledge about the technical tools and libraries used such that the concept can actually work in practice.
2. **Main**: Test the problem conditions live and see if the problem converged to finding an optimal policy for grouped scaling, with the right environment and agent formulation.
3. **Secondary**: Test if the right metrics and ideas taken from the papers presented in 2.5.3 were also appliable in this context.

4. **Secondary**: Provide for a reliable and precise monitoring support thanks to external tools.

Machine Learning and Reinforcement Learning Differences

Before introducing the concepts, a preliminary clarification has to be made.

Although for the whole work, we separated machine learning from reinforcement learning, it is more accurate to say that it actually is a branch of it. The crux lies in the fact that Machine Learning is usually taken as a synonym of "Supervised or Unsupervised Learning", especially in the papers we have discussed before. Reinforcement learning still involves using a massive amount of data to train models or neural networks (in deep-reinforcement learning). But the training and the final outcome are what really differs.

The main difference is that **Supervised** and **Unsupervised learning** take whole datasets as their input, while **Reinforcement Learning** takes live data to train. In reinforcement learning, the models that are being trained are used and updated in real time during training.

Hyperparameters

We will now quickly introduce hyperparameters. In both and more recently more in Deep Learning, hyperparameters play a big role. **Hyperparameters** are variables set at the beginning of each training session by an human and vastly determine how certain aspects of training are approached, like the speed of learning or the size of the Neural Networks. These are usually optimised by either studying the problem formally or (more commonly) by comparing various runs to see which fit best to the problem.

4.2 Reinforcement Learning Introduction and Key Concepts

The following dissertation is inspired by the reference book *"Reinforcement Learning: An Introduction" by Sutton and Barto* (49). Concepts from Deep Learning will also be drawn from (50).

*Reinforcement Learning (**RL**)* is a computational approach where agents learn to make decisions by interacting with an environment and receiving feedback, similar to how humans and animals learn from experience. Imagine a scenario where a robot learns to navigate a maze. If it reaches the end successfully, it receives a reward (positive signal), encouraging it to repeat this behavior. If it hits a wall, it receives a penalty (negative signal), discouraging that action. Unlike the trial-and-error learning in humans, RL algorithms systematically learn from this feedback to improve their actions over time.

Reinforcement Learning has a wide range of real-world applications across various domains:

- **Robotics**: RL is used to train robots to perform complex tasks, such as grasping objects, navigating environments, and interacting with humans.
- **Finance**: In financial markets, RL is used for algorithmic trading, portfolio management, and optimizing investment strategies.
- **Healthcare**: RL is applied in personalized medicine, optimizing treatment plans, and managing healthcare resources.

Formally, RL is modeled as a *Markov Decision Process (MDP)*, which involves sequential decision-making. An **MDP** is defined by:

- A set of **states** S,
- A set of **actions** A,
- A **transition function** $P(s_{t+1}|s_t, a_t)$ that describes the probability of moving from one state to another given an action,
- A **reward function** $R(s_t, a_t, s_{t+1})$ that assigns a numerical value to state transitions,
- A **discount factor** $\gamma \in [0, 1)$ that determines the present value of future rewards.

The goal in RL is to find a policy $\pi_\theta(s)$, which is a mapping from states to actions, that maximizes the **cumulative reward over time**, also known as the **expected return**. However, finding a globally optimal policy is challenging due to the complexity of the state-action space and the fact that the solution is not guaranteed to be globally optimal.

In **Reinforcement Learning**, at each time step t, an **Agent** interacts with an **Environment** by observing the **Current State** $s_t \in S$ and taking an **Action** $a_t \in A$. This action leads to a new state $s_{t+1} \in S$ and generates a **Reward** $r_t(s_t, a_t, s_{t+1}) \in \mathbb{R}^1$, which provides feedback on the action's effectiveness. The Agent updates its **Policy** $\pi_\theta(s)$ based on this feedback, where the policy represents the map function $a = \pi(s)$ **or** the probability $\pi(a|s)$ of taking action a in state s, and where θ are the **policy parameters**.

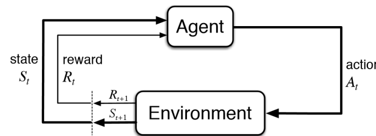


Figure 4.1: Reinforcement Learning Schema

4.2.1 The Agent's Objective

This section provides a concise mathematical overview of Reinforcement Learning to introduce key concepts relevant to our work.

The Agent's objective is to optimize its **Policy** $\pi_\theta(s)$ by estimating and maximizing the **State-Value Function** $V^\pi(s)$, which represents the expected return from state s under policy π. The State-Value Function is defined in:

$$V^\pi(s) = E_\pi\{R_t|s_t = s\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\} \quad (4.1)$$

Where:

- E_π denotes the expectation under policy π,

- γ is the discount factor, controlling the importance of future rewards.

The **Action-Value Function** $Q^{\pi}(s,a)$ extends this concept by considering both the state and the action taken and is shown in Equation 4.2:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\} \tag{4.2}$$

The optimal policy π^* is one that maximizes the Action-Value Function:

$$a = \text{argmax}_a Q^{\pi}(s,a) \tag{4.3}$$

The problem is solved when the value function $V^*(s)$ under policy π^* satisfies Equation 4.4:

$$V^*(s) = \max_{\pi} V^{\pi}(s), \forall s \in S \tag{4.4}$$

Methods to solve RL problems often involve optimizing both functions. One method frequently employed is **Generalized Advantage Estimation (GAE)** (51), which balances bias and variance in estimating the advantage function.

4.2.2 A Bit of Classification in RL: Value and Policy Based Methods

To solve RL problems, two main approaches are typically used: **Value-Based** and **Policy-Based Methods**.

Value-Based Methods

They focus on estimating the Action-Value Function $Q^{\pi}(s,a)$. One key approach here is the **Bellman Equation**, show in Equation 4.5:

$$Q^{\pi}(s_t,a_t) = E_{t+1}\left[r_{t+1} + \gamma Q^{\pi}(s_{t+1},a_{t+1})\right] \tag{4.5}$$

This equation can be solved **iteratively**, forming the basis of many RL algorithms. These methods can be further classified based on the assumptions they make (e.g., **Model-based** vs. **Model-free**) and the type of policy used during training (**On-Policy** vs. **Off-Policy**).

Policy-Based Methods

They directly seek to optimize the policy π^*, often using **Gradient-Based Methods**. The goal is to solve a **Gradient-Ascent Problem** on the objective function show in Equation 4.6:

$$\nabla_{\theta} J(\pi_{\theta}) = E\left[\nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi}(s,a)\right] \tag{4.6}$$

Where $\nabla_{\theta} \log \pi_{\theta}(a|s)$ is the gradient of the log-probability of taking action a in state s. This approach optimizes the policy parameters θ to maximize the expected return.

Actor-Critic Methods

Actor-Critic methods combine the strengths of both Value-Based and Policy-Based methods. In these approaches, an **Actor** updates the policy directly (Policy-Based), while a **Critic** evaluates the actions taken by the Actor using a Value-Based method. This combination allows for more efficient and stable learning. Deep-learning based algorithm that implement this method and are widely used in the field are *Deep Deterministic Policy Gradient* model (52) (**DDPG**) and *Proximal Policy Optimization* model (46) (PPO).

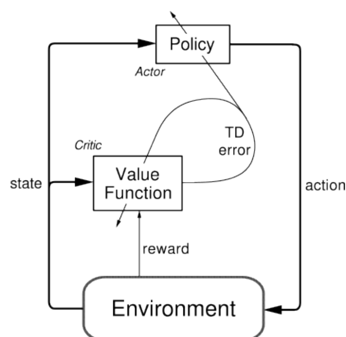


Figure 4.2: Actor-Critic schema

4.2.3 Agent Model Used: PPO

Proximal Policy Optimization (**PPO**)(46) is an Actor-Critic method and a safer alternative to other Actor-Critic models like Deep Deterministic Policy Gradient(52) (DDPG). PPO seeks to improve the policy while preventing it from deviating too far from the previous one, achieved through a **Clipping** mechanism in the objective function. In fact, this mechanism filters out the possible parameter θ updates that would produce a completely different policy from the current one. It is heavily inspired by the *Trust-Region Policy Optimization* model (53) (**TRPO**), which is another similar policy gradient algorithm.

The choice of **PPO** was influenced by its proven performance and stability, as demonstrated in (3), and its status as a widely researched and implemented model. The theoretical knowledge will not be presented in the scope of this work.

The PPO Agent is composed by a Critic Network and an Actor Network, in our case of the same dimension: **2 layers, 64 neurons each**.

4.2.4 The Concept of Exploration and Exploitation

A crucial aspect of RL is the balance between **Exploration** and **Exploitation**. **Exploration** involves trying new actions to discover potentially better strategies, while **exploitation** leverages known actions that yield high rewards. This trade-off is essential in RL because focusing too much on exploitation might prevent the agent from discovering better long-term strategies, while excessive exploration can slow down learning.

One common approach to managing this trade-off is the **epsilon-greedy strategy**, where the agent chooses a random action with probability ε (exploration) and the best-known action with probability $1-\varepsilon$ (exploitation). More sophisticated strategies involve adapting ε over time. In PPO, this difference is embedded in the agent's algorithm, so it is not controlled via a single hyperparameter as here.

4.2.5 RL model evaluation

There are various ways of evaluating a RL model and its training. The traditional way is monitoring the average returns. This is based on the definition we have given in 4.2.1.

Other approaches are possible: to evaluate a RL training session, one could monitor various time-series data. We will evaluate two as they will be the ones that we will use:

- **Average Rewards over the timesteps**: Directly monitoring the reward over the timesteps give us a **qualitative** and **quantitative** sensation of how the training is going. Depending on the model, the agent learns every t timesteps. At each of these intervals, if the training goes correctly, the rewards **should** go up and stabilize after a while in a specified range. That means that an optimal solution (local or global) is reached. In this case we say that the problem **has converged**. The amount of timesteps needed to see a change in this depends on the problem definition, the cardinality of the **Possible States**, the cardinality of the **Possible Actions** and many other environmental factors.

- **Entropy of Policy**: It is a measure that expresses the randomness and exploration factor of policy. We use it as in PPO there is no specific exploration vs exploitation parameter to be set, but it is specifically integrated in the model implementation. An entropy that tendentially goes down to 0 means that the agent is learning and stabilizing on a policy, additionally proving that it is converging to a solution.

An additional custom evaluation that we consider is the **Difference Action vs Old Best Replica**: these two different time-series data are representing the action taken at step t from the Agent (the one that was actually applied) compared to the action that a generic HPA would consider **for those metrics**, even if at a higher level of aggregation (from **microservices** to **groups**, those metrics still maintain the same meaning). With time if those two would be close it would mean that the reinforcement learning agent is actually learning to apply a scaling action similar to the HPA.

4.3 Proposal

The solution that is proposed to solve this second question is a fully customisable Reinforcement Learning Framework for Grouped Scaling. The solution implementation involved four big steps:

1. **The main one**: The creation of the environment; This meant defining all the **logic**, the **rewards**, the **states**, the **action** spaces and the various tools connections;

2. **Adaptation** and **improvement** on the code of the Agent's model from AWARE (3);

3. Creation of a **finely engineered reward** function that would account for more aspects to reward and to penalize comapred to previous solutions;

4. **Integration** with training monitoring tools.

This solution proposal is to be seen as the natural evolution of the **group scaler operator** presented in the previous chapter, inspired by the various papers seen in the first State of the Art section 2.5. Seen the two main objectives in 4.1, we wanted for the Agent to learn something that was easy to recognise in a test environment to see if it worked, while also learning a policy that converged easily. To do so, most of the work goes into the reward function, as it is the one that effectively guides the training. To test this, the rewards have been given based on the satisfaction of various objectives: among those, there was also the similarity between the proposed action and the HPA proposed action. This has been done to give a baseline for the Agent to learn in case all the other factors were not predominant enough. This will be seen in detail in the rewards algorithm in 4.4.1.

4.3.1 The Multi-Agent Grouped Scaler: Logic

The problem is approached by modelling it as a **multi-agent reinforcement learning problem** (**MARL**) (54). Multiple agents are initialized and tasked with learning an **optimal policy** for each group of microservices. The number of agents that is initialized is equal to the number of groups that are statically defined.

At the beginning, the environment and the various agents are initialized with all hyperparameters set to values decided in the arguments. Then the application is deployed and the workload generator is also initialized.

The training is composed of **episodes**. Each episode is composed of **timesteps**. In an episode the workload generator is activated with a random setting and then the episodes are executed with all the respective timesteps.

1: The environment fetches metrics from the metrics provider for some time using Prometheus.
2: Group and average the metrics **per time** first and then **per group**, generating the **states** for each agent.
3: **for** timestep t **do**
4:     **for** each agent **do**
5:         Take the current state for the group.
6:         Pass the information through the Neural Networks.
7:         Choose the scaling action with the highest probability (**Stochastic Policy**) for the associated group.
8: The environment receives the actions and sends the Kubernetes API requests to scale the microservices with the number of replicas requested.
9: Wait for the action to be completed.
10: Fetch new metrics for some time and create the new state, as in 1.
11: Compute rewards **for each agent** based on (s_t, a_t, s_{t+1}).
12: The new state becomes the s_t as the loop for timestep t ends
13: Return the rewards to the managing script.
14: The environment generates data for live monitoring.

After **n_ timesteps**, the states, rewards and actions taken are sent to the various agents, that subsequently apply the PPO algorithm to train the neural networks. The **episode** is therefore completed and this whole loop repeats for *n_* **episodes**. Data for monitoring is also sent at this time. Every now and then, the script saves logs of the trajectories, saves checkpoints of the neural networks so that they can be reused in the future and changes the workload settings to another configuration.

Figure 4.3 shows the schema of the whole procedure.

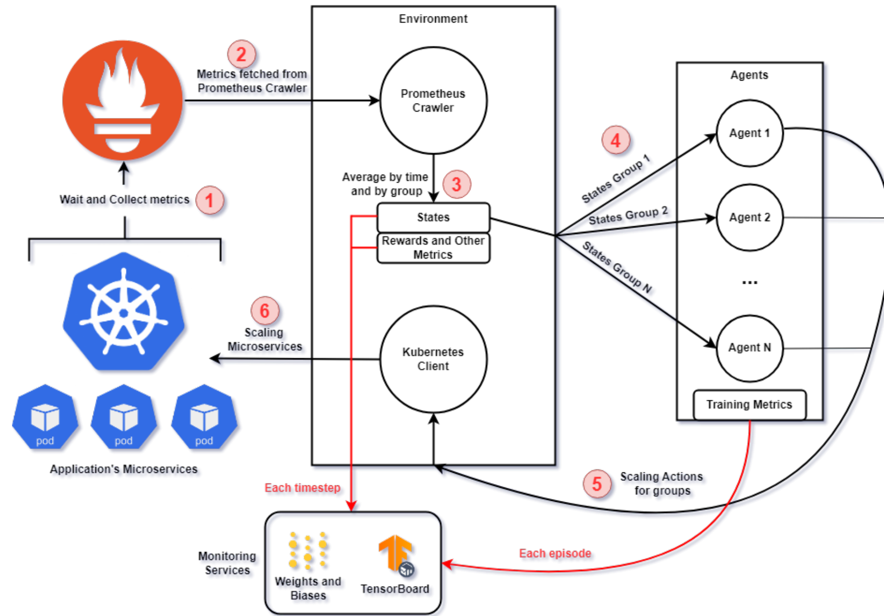


Figure 4.3: MARL Grouped Scaling schema

4.4 Multi-Agent Scaler Implementation

This work did not include any of the Operator knowledge acquired in the last chapter, as the complexity of development and debugging was not ideal in a context were multiple runs were needed like in this situation. So a normal Python script was deployed. A quick rally of the components will now be made and eventually how they have been implemented in the specifics.

4.4.1 Environment

The environment has been developed following OpenAI Gymnasium's (9) guidelines. Gymnasium is a standard API reference for Reinforcement Learning Environments programming. It provides a series of utilities to help guide and construct environments that are compatible with most of the Agents' implementations available on the web. Many communities have released

their different implementation for different environments, so it is beneficial to have common ground for comparison and testing while also being reliable and easy to understand, as it is documented and used by many developers.

Gymnasium provides a framework to develop single agent Reinforcement Learning experiments, but not for multi agent ones. There is another library derived from Gymnasium that addresses that: PettingZoo (10). This library provides a pair of APIs that lets the developer design and build environments in which the agents take actions in a sequential order during a turn (AEC API) or all in parallel during the same turn (Parallel API). Since the second option was the most congenial to our case, where multiple agents are asked by the environment to all take an action at the same time.

Metrics

Metrics are retrieved by exploiting Prometheus(7) and with additional metrics added by Istio(8), presented in 2.3.3 and 2.4.1.

For each microservice the retrieved metrics are:

- $CPU_Util_{ms} = avg_{ms}\frac{cpu_util_{pod}}{cpu_limit_{pod}}$, which represents the average of the CPU Utilization among the Pods associated to a Microservice.

- $MEM_Util_{ms} = avg_{ms}\frac{mem_util_{pod}}{mem_limit_{pod}}$, which represents the average of the Memory Utilization among the Pods associated to a Microservice.

- $num_replicas$

then for the entire application the retrieved metrics are:

- $latency_SP_{50} = \frac{0.50Quantile(request_latency)}{SLA_latency_50}$

- $latency_SP_{95} = \frac{0.95Quantile(request_latency)}{SLA_latency_95}$

These first two metrics are a reiteration of the ones seen in 2.3.3 for the resource utilization, while the last two ones are a reiteration of the ones presented in 2.4.1 for the latencies quantiles, already mixed with the SLA preservance ratio concept seen while presenting FIRM(2) and AWARE(3) in 2.5.3. This last value indicates how much the SLA is respected (1 totally respected, values toward 0 less and less respected). These metrics have been heavily inspired by the ones used in (2) and (3), even if used in a different manner. The SLA limits for them to be considered respected are decided at the user level and defined directly in the metric and state definition. For SLA, base values of 20 ms for **latency50** and 5000 ms for **latency95** were used.

These metrics are taken for some time: in our case $3seconds * num_tries$, where the 3 seconds is the interval between a metric update and another one. The metrics are then averaged on this interval. This is to prevent that the state contains spikes which are too high or too low and that are not reflecting on the cluster's true state, while also letting the application adapt to the new replica count and distribute the load well. Usual values for num_tries is between 4 and 10. This value alone determines the time it takes to train, as seen later in 4.5. These metrics are then to the environment which will create the states.

Objective **4.** presented in 4.1 has then been addressed.

States

The states are defined through Gymnasium's **Spaces** classes. The library gives a lot of options to create custom spaces with preset boundaries, useful for states and action spaces definition.

Since there are multiple agents that each manage a different group, each agent will need a vector containing all the states associated to its group. Once the metric are fetched per-microservice and averaged over time, the per-group states that have been used for the environment definition were the following:

$$[avg_cpu_util, avg_cpu_util, num_replicas, latency_S P_{50}, latency_S P_{99}]$$

with the following boundaries:

$$[[0,2],[0,2],[1,limit_{group}],[0,1],[0,1]]$$

Particularly for the first two, these are directly taken from the definition that is given in 3.1.1.

Actions

Here too Gymnasium's **Spaces** classes have been used. For the actions there were two ways of approaching the problem. Either defining it as an **Increment/Decrement** action i.e. taking the current replicas and performing a **Relative** action based on it, or taking values in a range and performing an **Absolute** action. The former implied more difficulties with the knowledge I had at the beginning, so the latter approach was chosen.

Therefore, for each group a static limit was imposed based on performance measurements. The action is:

$$[num_replicas]$$

in the following boundary:

$$[1, limit_{group}]$$

Rewards

We recall the following reward formula in Equation 4.7, taken from FIRM(2) and AWARE(3):

$$r_t = \alpha * SP_t * |R| + (1-\alpha) * \sum_{i=0}^{|R|} \frac{RU_t}{RLT} \tag{4.7}$$

where **SP** is the **SLA Preservance Ratio**, $|R|$ is the number of resources, *RU* the various Resources utilizations (CPU, Memory...) and **RLT** their limits in the pods. With the number of states that they defined this reward function has given its good results.

In our case, however, I deemed this neither sufficient nor expressive enough. With such a small state space, I wanted to give my personal contribution by creating a more finely engineered reward function, that would account for more aspects.The reward function takes the previous state, the action that has been performed and the new state.

The differences between my version and the one proposed in the previous two mentioned papers are on the resource usages, the penalty and the bonuses and in general the whole objective. Here is a summary of what it accounts for:

Algorithm 1 Compute Reward r_t

1: **Input:** s_{t-1}, a_t, s_t
2: **Output:** $r_t \in (-1, 1)$
3: **Given:** $CPU_Util_{Desired}$, $MEM_Util_{Desired}$
4: **BestOld_ HPA_ Replica** $\leftarrow$ max $\left(\frac{CPU_util_{t-1}}{CPU_Util_{Desired}}, \frac{MEM_util_{t-1}}{MEM_Util_{Desired}}\right)$
5: **TypeOfScale** $\leftarrow (a_t$ or $num_replicas_t) - num_replicas_{t-1}$
6: **DifferenceActionDesired** $\leftarrow (a_t$ or $num_replicas_t) -$ **BestOld_ HPA_ Replica**
7: $\alpha_{resources} \in [0,1]$ ▷ Weight between instantaneous performance and SLA, typically 0.7
8: $\beta_{cpu} \leftarrow \begin{cases} 0.7 & \text{if } \textbf{BestOld_HPA_Replica} == \text{CPU predominates,} \\ 0.3 & \text{if } \textbf{BestOld_HPA_Replica} == \text{MEM predominates} \end{cases}$
9: $score_{cpu} \leftarrow 0.5 \cdot \beta PDF(CPU_Util_t) + 0.5 \cdot \beta PDF(CPU_Util_{t-1}) * \delta_{cpu}$
10: $score_{mem} \leftarrow 0.5 \cdot \beta PDF(CPU_Util_t) + 0.5 \cdot \beta PDF(CPU_Util_{t-1}) * \delta_{mem}$
11: δ_{cpu} and δmem decided by another algorithm (not seen here). βPDF is a negative skew function peaking at 1 in 0.8
12: $penalty_{scaling} \leftarrow 0.15 \cdot \tanh(-$**DifferenceActionDesired**$)$
13: $bonusCPU_{scaling} \in [-0.2, 0.3]$ if $CPU_Util_t - CPU_Util_{t-1}$ is small and $CPU_Util_t < CPU_Util_{Desired}$
14: $bonusMEM_{scaling} \in [-0.2, 0.3]$ if $MEM_Util_t - MEM_Util_{t-1}$ is small and $MEM_Util_t < MEM_Util_{Desired}$
15: $\gamma_{latency_{50}} \in [0,1]$ ▷ Weight for median latency in SLA, typically 0.5
16: **WE GET:**
17: $r_t \leftarrow \alpha_{resources} \cdot (\beta_{cpu} \cdot score_{cpu} + (1 - \beta_{cpu}) \cdot score_{mem}) + (1 - \alpha_{resources}) \cdot (\gamma_{latency_{50}} \cdot SP_{latency_{50}} + (1 - \gamma_{latency_{50}}) \cdot SP_{latency_{95}})$
18: $r_t \leftarrow r_t + penalty_{scaling} + bonusCPU_{scaling} + bonusMEM_{scaling}$
19: $r_t \leftarrow tanh(r_t)$, for normalizing in $[-1,1]$

1. Give first a general score based on the current and last CPU and Memory Utilizations and the respecting of the SLA limits. The score tried to account for the actual effectiveness of scaling and if the resource usage was close enough to the target resource utilisation. That is why βPDF was used, as it was defined as a negative skew function peaking at 1 in 0.8: The closer the utilizations are to that point, the better the score. If the utilisation is over 0.8, the score goes down to 0 pretty quickly, giving a worse score. This is done to promote the usage of the resources to a good level and to promote less situations of under-usage or throttling.

2. Assign a penalty based on scaling **"too much"** compared to the previous state, moved by the reason that scaling takes time and if it is not actually needed it should not be enforced.

3. Assign some bonuses if the metrics are stable enough between a state and the other.

4. Normalize the whole reward using a *tanh* function. This is done to ensure that rewards are assigned in a space that is $\in [-1,1]$. This gives stability properties to the algorithm.

Especially the first point explains the behaviour I wanted to start from: the one of the **HPA**. In fact, even if in a more refined matter, **1.** says that a higher score is assigned if the resource utilization among the two present is close to a desired value, which is set to 0.8. Since the HPA works by scaling to as many replicas as the desired resource utilization of the most used resource, the total resource score will get to a higher values if the whole conditions are respected. The other factors add complexity and expressivity to the function, but as we will see in 4.5, this behavior is eventually learnt by some agents.

Kubernetes Client

As also presented in the Operator, the connection to the Kubernetes Client just involves reading the local **kubeconfig** file. The configuration is then stored locally from the Kubernetes client, which can be called from any point of the code to either query the cluster or send requests to it. It is really simple and its usage was facilitated by all the experience done on the Operator in the previous chapter 3.3

4.4.2 Agents

The agent, as stated in 4.2.3, is adapted from AWARE's version proposed in their artifact (3). Their implementation presented some things that were not useful to our context, as also a not clear separation between the Agents' and the Environment's logics.

The implementation is made with PyTorch (55). The size of the two Neural Networks can be decided at the beginning if no checkpoint is requested. In this case what can be decided is the number of layers and the number of neurons inside a layer.

4.4.3 Workload Generator

The workload generator is a personal contribution. I created a Locust (28) client that takes a *locustfile* to generate the typical user pattern of interaction with the application (like logging in, browsing the teas and buying some) and generate a random amount of users (from 10 to 150) over a random amount of time (from 30 to 300 seconds) with different user generation rates (how many users are generated every second until the maximum amount of users is reached, goes from 1 to 10). The Workload can be executed in 3 different profiles, all different in the amount of seconds that each users waits between a task and another, chosen at random at runtime: **Low** (5s), **Medium** (between 1-3s), **High** (1s).

This workload runs on a separate thread than the script for the training and generates a new workload when the old workload is not running anymore. **Future works** could make this better and more variegated, to better "guide" the PPO Agents by generating more diversified states.

4.4.4 Monitoring

Figure 4.4: Tensorboard and WandB

For monitoring, two tools have been implemented and various metrics are continuously scraped from the training thanks to them. The tools are "*Tensorboard*"(11) and "*Weights and Biases*"(12) (**WandB**). These two tools let the developer gather a series of time-series data and construct dashboard with graphics generated from them.

4.5 Experiments

As stated in 4.1, there are 4 goals. In 4.4.1 in the rewards section, we anticipated the behaviour that we wanted for our Agents to learn, implying that it would sometimes tend to act like an HPA but accounting for more aspects. By proving that the agents "converge" to a similar behavior we want to prove that **grouped intelligent scaling works**. and that **its associated reinforcement learning problem converges**. We will just present a quick draft of a possible optimal solution, as more work and more training time is definitely needed for this kind of task.

Let us start with the configurations.

4.5.1 External configurations: Nodes and Groups

The experiments have been run on a large set of clusters of GRID'5000. The machines that have been used have mainly been **chirop** (for longer trainings), **chifflot**, **chiclet** and **nova**. Many runs have been taken and some of them have been chosen to be shown in this context. However, as we will see, the training sessions have all converged to an optimal solution (which is not known if it is global or local) in a relatively quick time, regardless of the configuration of the machines and the nodes that are used. This, along with other reasons we will show later, already proves that the framework I proposed is **applicable anywhere** in the GRID'5000 cluster and possibly, with due refinement, in a general cluster that deploys Kubernetes and a generic application.

Application and Groups

The application used was yet again TeaStore (22), for its simplicity of deployment.

The objective in this round of experiments was to scale all the application by training to scale the microservices that actually needed that. For that, again I took inspiration from the result shown in Figure 3.2.

The groups were then subdivided in this schema:

- Group 0 $\rightarrow$ Agent 0 = [**teastore-webui**], limit $\in$ [7,10]
- Group 1 $\rightarrow$ Agent 1 = [**teastore-image**, **teastore-auth**, **teastore-persistence**], limit $\in$ [3,6]
- Group 2 $\rightarrow$ Agent 2 = [**teastore-recommender**], limit $\in$ [1,3]

The limits for the replicas have been tuned around the various runs. These values are for sure **adequate** but probably **not the best**. Future works could account this aspect.

4.5.2 Internal Configurations: Hyperparameters

The tables below list the hyperparameters and environmental parameters used in the experiment, along with their roles. Most of the values for the hyperparameters have been adapted from other projects that implement the same agent model, **PPO**. A brief explanation of what each hyperparameter does is given, but all the implications that these have are not explained, as this part was not extensively studied by me.

Hyperparameter	Value	Description and Role
Learning rate	0.0003 ± 0.0001	Controls the step size in the optimization process. Lower values can lead to more stable but slower convergence.
Gamma (Discount factor)	0.99	Determines how much future rewards are discounted. Higher values emphasize long-term rewards.
K epochs	5	Number of epochs to update the policy in each iteration.
Eps clip	0.2	Clipping parameter for the PPO objective function to prevent large policy updates.
VF coefficient	0.5	Coefficient for the value function loss term, balancing the importance of value prediction.
Entropy coefficient	0.01	Encourages exploration by adding entropy to the loss function.
Max grad norm	0.5	Limits the gradient norm to prevent excessive updates and maintain stability during training.
GAE lambda	0.95	Parameter for Generalized Advantage Estimation (GAE), controlling the bias-variance trade-off.
Hidden size	64	Number of units in the hidden layers of the neural network, affecting model capacity.
Num layers	2	Number of layers in the neural network, affecting the complexity of the model.
Critic loss discount	0.05	Discount applied to the critic loss, balancing its contribution relative to other loss terms.
Minibatch size	5	Number of samples per minibatch, used for gradient descent updates.

Table 4.1: Hyperparameters for PPO Training

Parameter	Value	Description and Role
Total episodes	300, max 120 reached	Total number of episodes in the training process. Determines the amount of training experience.
Max steps per episode	50, 60, 100	Maximum number of steps allowed per episode, limiting the length of each training instance.
Cluster stability checks	[4 to 8]	Number of rounds to take metrics from Prometheus, **highly affects training time**.
Seed	39, 42, 561	Seed values for random number generation, ensuring reproducibility of results.

Table 4.2: Environmental and Training Parameters

The problem, as we will see from the results, converges within these configurations, suggesting that the chosen hyperparameters are effective for this scenario. Although there may be more optimal values, the focus here is on achieving a functional setup that serves as a foundation for future works.

Notably, the number of steps per episode is relatively low compared to more complex environments, such as those found in Atari games which can be in the order of 10^4 to 10^5(9)(10). However, this smaller number could be suitable given the simplicity of the environment and the limited range of possible actions. This reduced complexity allows agents to train **easily** with the policy-gradient mechanism, which is what the PPO algorithm is based on.

This configuration, therefore, offers a balance between computational efficiency and performance, providing a solid basis for further experimentation and refinement.

4.5.3 Runs and Evaluation

Of the numerous runs that have been conducted, 5 have been selected for their overall performance. They will be from now on referenced with run from **A.** to **E.**. The configuration is shown in Table 4.3:

Label	Runtime	Stability Checks	Steps/Episode	Seed	CPU Cores	Memory (GiB)	Hardware
A	61 hrs	8	100	42	1	5	chirop
B	39 hrs	4	100	39	2	5	nova
C	21 hrs	3	50	42	4	10	chiclet
D	14 hrs	5	60	561	4	10	nova
E	58 hrs	5	60	561	2	8	chiclet

Table 4.3: Summary of Experiments

One thing that has to be noted: even with similar or even equal parameters and hyperparameters, the different runs have not been equal. This is due to the **high variability of the environment and application usage**, given by the random workloads used and the different behavior of the application and the cluster resources, and the **high variability of the reinforcement learning process**, which can be highly influenced by the initial conditions and states and could potentially never converge. Fortunately, as we will see, this last thing does not happen in our case.

As visible from Figure 4.5, all the runs have stopped at a different amount of steps. This has to do with the time total time time of training, which takes into account the **number of calls for the averaged metrics** (controlled by the **Cluster stability checks** parameter) and the possible runtime guaranteed by GRID'5000 jobs. The training time of neural networks can be considered **negligible** in comparison to that, as they are very small.

The plots shown are made thanks to the use of the**Weights and Biases** monitoring tool. As visible in the plots, the more visible time series are a smoothed-out version of the original ones, as without it the results are not readable. Smoothing lets us notice and appreciate tendencies for the various time-series, especially for rewards. The smoothing setting that has been used was **Time Weighted EMA (exponential moving average)** with a smoothing factor of 0.9.

General Evaluation: Rewards

Figure 4.5 shows the **timesteps** on the X-Axis and the value of the corresponding reward on the Y-Axis. For the various runs, the rewards, throughout all steps (1 step = 1 action taken), tend to increase to certain values for all the various agents, no matter the amount of timesteps in

an episode/iteration. In these different runs, the number of steps in an episode changes, so the learning step is more or less distributed, but is not visible here. But it is possible to appreciate how, no matter how big the episodes are, the runs yield growing and more steady reward with time.

For example, for **D.**, the Agent 0 converges pretty quickly to an average smoothed reward value of around 0.8 in around 2000 timesteps, while for **E.** it converges to around 0.6 and stays there. This means that for the latter, the policy has shifted towards local optima given some different initial conditions. **Agent 0** definitely shows the most different behaviour among the various runs. This can be due to various reasons:

It also worth to note how **Agent 1** and **Agent 2** have a general better **convergence** than **Agent 0** among **all runs**. In my opinion, this can be most likely related to the fact that **Group 0** manages the least predictable microservice, which is the one that also receives the request: **teastore-webui**. This might mean that **Agent 0** either might:

- Needs more time to train (and surely more diversified workloads);
- Have converged to a local optima and it might not get out of it without more exploration.
- Not have optimal replicas limits.
- Be more influenced by other factors in the computation of rewards, effectively creating a different policy that is probably influenced more by the various fluctuations of the scores.

This last point could also explain why the other two Agents are converging "better", combined to the results we will see in Figure 4.8. In fact, **Agent 1** and **2** tend to converge pretty quickly and at the same pace to a pretty high reward of 0.8 in around 500 to 1000 timesteps for almost all runs. This may be due to the fact that the microservices included in those groups have a more **predictable** state and generally yield better rewards pretty early on.

Out of all the runs, one in particular was conducted by completely changing the reward function. This implemented a different version of the reward function proposed in the reward section in 4.4.1, and as seen it converges to a lower optima while still considering the same kind of behaviour.

These results then show how **it converges** and **it does so to an optimal value of rewards, which are defined in** $\in [-1,1]$.

(a) Agent 0 **(b)** Agent 1 **(c)** Agent 2

Figure 4.5: Rewards for the various runs throughout all the steps

Training Evaluation: Rewards and Entropy

Figure 4.6 shows some of the metrics that have been collected after each episode, which means at each **learning** phase of the various agents. The X-Axis represents the *i*-th episode, while the Y-Axis respectively the Average Rewards and Entropies among all agents, showing all runs. These metrics considered have been explained in 4.2.5 and are are a result of being averaged among the various Agents, to generally show the tendency of the whole training process. As visible:

- the **Rewards** are generally going up and all **tend to converge** to a value that is more or less optimal for the given condition; for most of the runs the values of the reward are > 0.5 and the tendency is up-going.
- The **Entropy** is going down, meaning the agents are all tending to explore less and actually use their policy more, effectively telling us that the rewards are actually being mapped to the *(state, action)* space. For runs **B.**, **C.** and **D.** it converged to almost 0 too quickly, signaling that the agents **may** converged too quickly and could have **either** used some more exploration or that the states that they were facing were always similar to one another, reinforcing the exploitation. This is unfortunately random and given by the variability of the experiment and the states. For run **E.** it converged later, potentially meaning that all the agents have explored for more time the *(state, action)* space

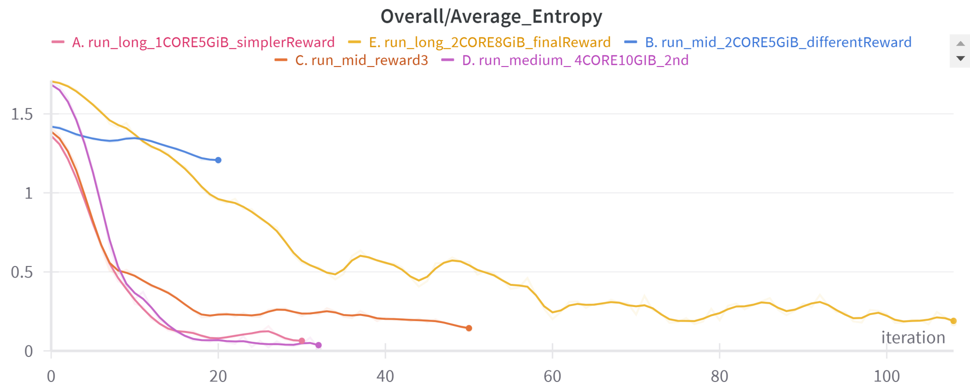
Analysis of run E. : States and Policy

We now focus on the run that had the most amount of steps: run **E.** It run with 50 **steps** per episode, for a total amount of 109 **episodes/iterations**, done in 57 hours of runtime.

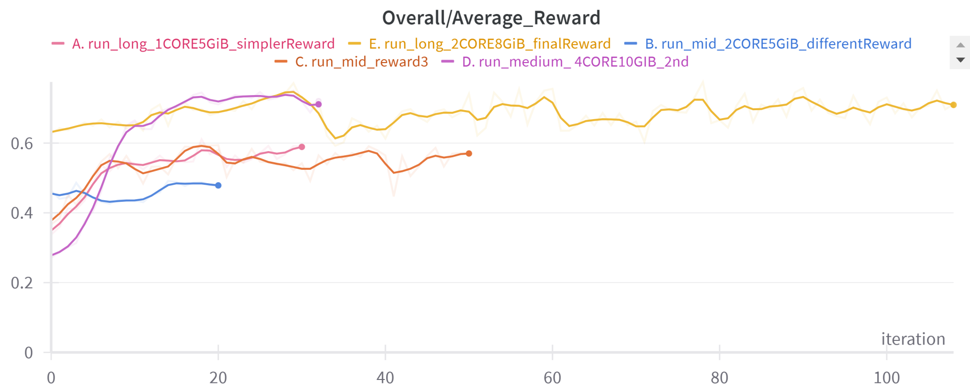
Figure 4.7 show he graphics for the various monitored states along the timesteps on the X-Axis and the various ratios representing the different states on the Y-Axis, and by so they don't have a unit of measurement. As visible, the smoothed states show that the agents succeed in generally maintaining a good value of said statistics, maintaining a good level of CPU and Memory utilization while trying to maximize the SLA maintenance, for each respective agent in each row. This proves that grouped scaling is possible with reinforcement learning and ensures good resource utilization and SLA maintenance if setup correctly. Further knowledge cannot be taken from this as of this work's study. Possible future studies could address this.

Finally, we analyze Figure 4.8, which show the behaviour described of how the Agents policies' decide over the various steps in the episodes, compared to the value of the replica that an eventual HPA would give using the CPU and Memory metrics for the group. On the X-Axis we find the timesteps and on the Y-Axis we find the two timeseries: A continuous line for the number of replicas decided by the agents' policies (and then applied) and a dashed line for the HPA recommendation for the number of replicas given by the corresponding states.

For **Agent 1** and **2**, the policy moves towards the same behavior as an HPA, meaning first of all that the agents are actually training a policy and that in this case it is training to become similar to the one imposed by the HPA, meaning that the reward function proposed in 4.4.1 actually accounted for that behaviour, and confirmed us that with enough engineering a good reward function is possible and can account for something more than single HPAs, like SLAs, penalties and bonuses. For **Agent 0** however, the tendency is less visible, and it is probably due



(a) Entropy



(b) Average Reward

Figure 4.6: Training Metrics among Agents

to the reasons stated in 4.5.3: the agent is most likely either creating a new policy that differs from the HPA one to try and that converges to a local optimum.

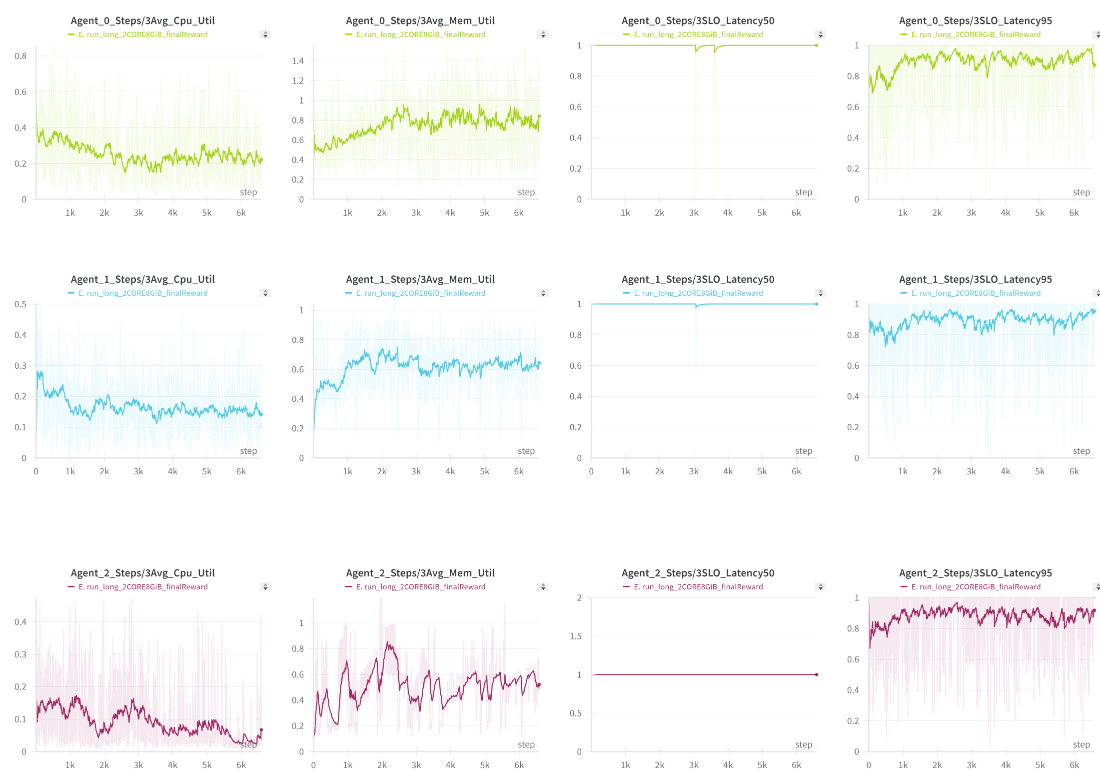
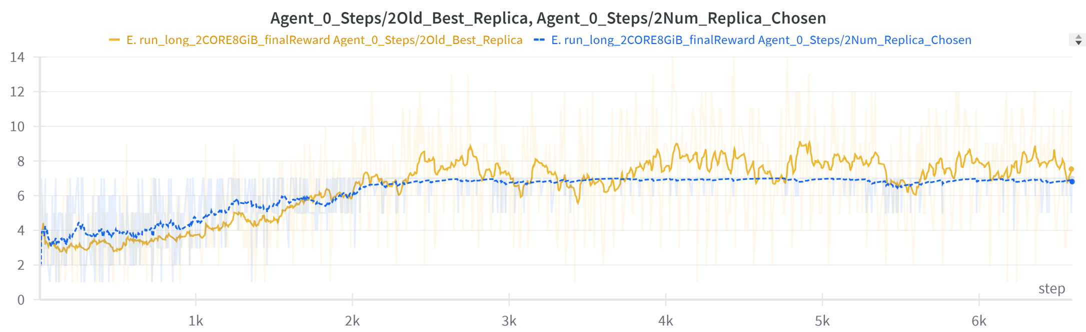
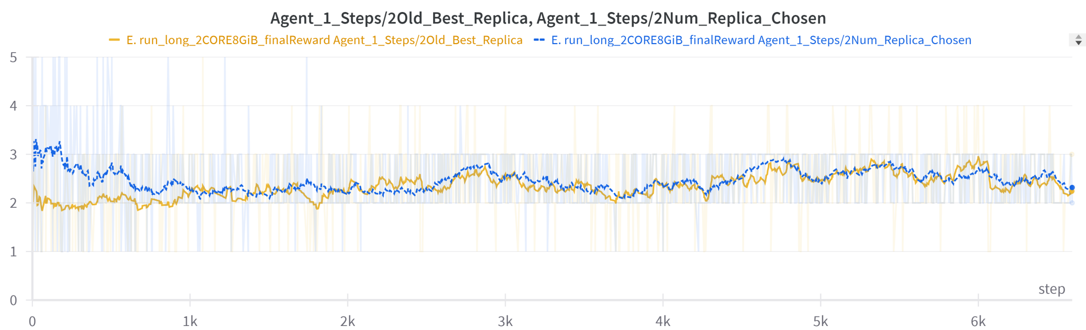


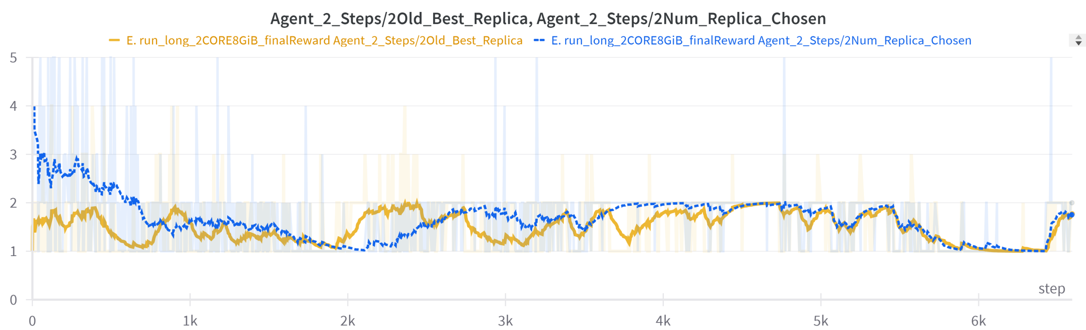
Figure 4.7: States of run E. per Group



(a) Agent 0



(b) Agent 1



(c) Agent 2

Figure 4.8: Difference of behaviours between Agents and respective hypothetical HPA best replicas

Chapter 5

Conclusion

In this report, we walked through the various concepts that were of matter during my work at LIG. All the concepts have been explained and then the two contributions to the SCALER (14) projects have been thoroughly presented. As seen from the previous chapters, from the first one I have produced:

- A customizable **Operator** left as an artifact that is usable from future developers to continue the study in this field.
- Some insights about horizontal scaling times and how they can affect future work.
- Experimentation setups to reproduce the same context on the French National Cluster of GRID'5000.

We have shown with this that grouped scaling works, that it is technically feasible thanks to the usage of a controller and that possible results can be extracted from its usage. The results of the work on the operator have been more experimental, without the actual objective of creating something innovative or that worked perfectly. We succeeded on that front. Despite all of this, grouped scaling, the Operator and the Insights have all been presented at COMPAS'2024 in Nantes, a conference about Parallelism, Distributed Systems and Architectures.

From the second one I have produced:

- A fully customizable Multi-Agent Reinforcement Learning Framework specifically tailored to the problem. The design included the Open-AI's Gymnasium compliant **environment** and **PPO agent implementation**.
- Some results that the training actually works and converges to a solution. It is not sure that the solution is globally optimal though, as this problem requires more guidance to fully explore the (state,action) space presented by the problem.
- A reward function that tries to push the agents to work like an HPA but with an enhanced state comparison.
- A simple yet fully customisable working general workload generator for whatever application is needed with Locust.

- A Python Prometheus Client extension that is used in this context but that is possible to use in other contexts.
- A support for monitoring live metrics as the training progresses, making the training experience easier significantly easier for the developer, compared to how State-of-the-Art artifacts handled this aspect.

With the experiments seen in this part, we have established that a **reinforcement learning** solution to the **grouped scaling problem** is for sure possible. With the right ideas and experiments, we have shown that the agents can converge and in some cases even converge to solutions that yield high rewards (especially for those groups of microservices that are less requiring). Future works should definitely study: better rewards functions, more states to insert into the problem, integrate vertical scaling along with horizontal one, longer trainings setups and re-usage of the trained models.

All the artifacts are left in the team's GitLab.

Bibliography

[1] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, and R. Buyya, “Machine learning-based orchestration of containers: A taxonomy and future directions,” *ACM Comput. Surv.*, vol. 54, sep 2022.

[2] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, “FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 805–825, USENIX Association, Nov. 2020.

[3] H. Qiu, W. Mao, C. Wang, H. Franke, A. Youssef, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, “AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems,” in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, (Boston, MA), pp. 387–402, USENIX Association, July 2023.

[4] “Kubernetes Documentation.” https://kubernetes.io/docs/home/, 2023. [Online; accessed 13-Feb-2024].

[5] “Kopf Operator Framework Repository on GitHub.” https://github.com/nolar/kopf, 2024.

[6] “Autoscalers Repository on GitHub.” https://github.com/kubernetes/autoscaler, 2024.

[7] “Prometheus Documentation.” https://prometheus.io/docs/introduction/overview/, 2024. [Online; accessed 6-Mar-2024].

[8] “Istio Documentation.” https://istio.io/latest/docs/, 2024. [Online; accessed 11-Apr-2024].

[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” 2016.

[10] J. K. Terry, B. Black, N. Grammel, M. Jayakumar, A. Hari, R. Sullivan, L. Santos, R. Perez, C. Horsch, C. Dieffendahl, N. L. Williams, Y. Lokesh, and P. Ravi, “Pettingzoo: Gym for multi-agent reinforcement learning,” 2021.

[11] “Tensorboard visualization toolkit.” https://www.tensorflow.org/tensorboard. [Online; accessed 20-Aug-2024].

[12] “Weights and biases.” https://wandb.ai/. [Online; accessed 20-Aug-2024].

[13] “GRID5000.” `https://www.grid5000.fr/w/Grid5000:Home`, 2024.

[14] "SCALER: Smart **SCAL**ing for Micros**E**rvice A**R**chitectures." https://scaler.gricad-pages.univ-grenoble-alpes.fr/web/, 2024.

[15] https://erods.liglab.fr/, 2024.

[16] M. Yan, X. Liang, Z. Lu, J. Wu, and W. Zhang, "Hansel: Adaptive horizontal scaling of microservices using bi-lstm," *Applied Soft Computing*, vol. 105, pp. 107–216, 2021.

[17] "Software architecture: a definition by the sei." https://www.sei.cmu.edu/our-work/software-architecture/. [Online; accessed 23-Jul-2024].

[18] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A rapid review," in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pp. 1–7, 2019.

[19] "Docker: Definition of containers." https://www.docker.com/resources/what-container/. [Online; accessed 23-Jul-2024].

[20] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[21] "Kubernetes: Definition of pods." https://kubernetes.io/docs/concepts/workloads/pods/. [Online; accessed 30-Jul-2024].

[22] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "Teastore: A micro-service reference application for benchmarking, modeling and resource management research," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 223–236, 2018.

[23] "Vpa github." https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler. [Online; accessed 30-Jul-2024].

[24] S. N. A. Jawaddi, M. H. Johari, and A. Ismail, "A review of microservices autoscaling with formal verification perspective," *Software: Practice and Experience*, vol. 52, no. 11, pp. 2476–2495, 2022.

[25] "Kubernetes metrics pipeline." https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/. [Online; accessed 14-Aug-2024].

[26] "Kubernetes prometheus stack." https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack. [Online; accessed 18-Aug-2024].

[27] "cadvisor metric collector." https://github.com/google/cadvisor. [Online; accessed 24-Aug-2024].

[28] "Locust: An open source load testing tool." https://locust.io/. [Online; accessed 10-Aug-2024].

[29] "Jaeger Documentation." https://www.jaegertracing.io/docs/1.56/, 2024. [Online; accessed 11-Apr-2024].

[30] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, (New York, NY, USA), pp. 412–426, Association for Computing Machinery, 2021.

[31] "Ieee internet of things journal." https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=6488907. [Online; accessed 10-Aug-2024].

[32] "Usenix conferences." https://www.usenix.org/conferences. [Online; accessed 10-Aug-2024].

[33] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), pp. 3–18, Association for Computing Machinery, 2019.

[34] "Sock shop : A microservice demo application." https://github.com/microservices-demo/microservices-demo. [Online; accessed 10-Aug-2024].

[35] H. Ahmad, C. Treude, M. Wagner, and C. Szabo, "Smart hpa: A resource-efficient horizontal pod auto-scaler for microservice architectures," *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, pp. 46–57, 2024.

[36] "Online boutique : A googlecloudplatform microservice demo application." https://github.com/GoogleCloudPlatform/microservices-demo. [Online; accessed 10-Aug-2024].

[37] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *2019 IEEE International Conference on Web Services (ICWS)*, pp. 68–75, July 2019.

[38] "Locust: An open source load testing tool." https://github.com/lightstep/hipster-shop. [Online; accessed 10-Aug-2024].

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), pp. 6000–6010, Curran Associates Inc., 2017.

[40] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1994–2004, July 2019.

[41] M. Wajahat, A. Karve, A. Kochut, and A. Gandhi, "Mlscale: A machine learning based application-agnostic autoscaler," *Sustainable Computing: Informatics and Systems*, vol. 22, pp. 287–299, 2019.

[42] M. Xu, C. Song, S. Ilager, S. S. Gill, J. Zhao, K. Ye, and C. Xu, "Coscal: Multifaceted scaling of microservices with reinforcement learning," *IEEE Transactions on Network and Service Management*, vol. 19, pp. 3995–4009, Dec 2022.

[43] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 80–90, July 2019.

[44] "Train ticket: A benchmark microservice system." https://github.com/FudanSELab/train-ticket. [Online; accessed 10-Aug-2024].

[45] "Gitlab depend." https://gitlab.engr.illinois.edu/DEPEND. [Online; accessed 10-Aug-2024].

[46] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[47] "Kubernetes operator pattern." https://kubernetes.io/docs/concepts/extend-kubernetes/operator/. [Online; accessed 10-Aug-2024].

[48] "Kubernetes rbac." https://kubernetes.io/docs/reference/access-authn-authz/rbac/. [Online; accessed 10-Aug-2024].

[49] R. S. Sutton and A. G. Barto, *Reinforcement Learning: an Introduction*. The MIT Press, 2018.

[50] X. Wang, S. Wang, X. Liang, D. Zhao, J. Huang, X. Xu, B. Dai, and Q. Miao, "Deep reinforcement learning: A survey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, pp. 5064–5078, April 2024.

[51] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2018.

[52] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019.

[53] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," 2017.

[54] L. Busoniu, R. Babuska, and B. De Schutter, "A comprehensive survey of multiagent reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, pp. 156–172, March 2008.

[55] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.