POLITECNICO DI TORINO

Master's Degree course in Computer Engineering, Computer Networks and Cloud Computing

Master's Degree Thesis

Scheduling Kubernetes Tasks with Reinforcement Learning

Supervisors

Prof. Alessio SACCO

Prof. Marchetto GUIDO

Candidate

Sonia MATRANGA

July 2024

Abstract

In the world of cloud services, the growing complexity of distributed applications and the increase in energy consumption necessitate more efficient management of resources. For this reason, orchestrators such as Kubernetes are widely employed to automate the handling of workloads and resource usage, determining moment by moment the most suitable node on which to start a new task. On the other hand, the expanding application of artificial intelligence algorithms, particularly reinforcement learning, opens up new development opportunities. These advancements allow the creation of increasingly autonomous and state-of-the-art systems.

This thesis introduces and develops a different approach to scheduling within Kubernetes clusters. Specifically, the proposed scheduler utilizes a Deep Q-Network (DQN) reinforcement-learning algorithm, integrating a custom plugin in the scheduling chain's scoring phase to optimize the distribution of load across available nodes. In developing this innovative and intelligent approach, each RL model has been trained to learn a distinct policy with specific objectives such as load balancing, energy consumption optimization, or node-user latency optimization.

The reinforcement-learning algorithm implemented in the plugin dynamically assesses the resources available on cluster nodes and learns to manage them while adhering to user-defined constraints. By assigning a score to each node based on its suitability for hosting new pods, this intelligent approach supports decision-making and serves as a predictive tool for the scheduling system. Over time, this enables the system to continually improve its decisions regarding the optimal distribution of new workloads, in accordance with the learned policy.

The implementation has been tested on a Kubernetes Kind environment, allowing for an assessment of the overall performance of the developed system and the effectiveness of the proposed approach. In particular, results shows that our policy, referred to as *EC-RL*, learned by the agent, proves to be the best choice when the goal is to reduce energy consumption and node-user latency, both compared to the other tested policies and to the default behavior of the Kubernetes scheduler.

Acknowledgements

To my supervisor, **Alessio Sacco**, thank you for guiding me throughout this long journey and for always supporting and encouraging me to give my best. Thank you for always being there for me with the right words at the right time.

I would like to thank **Doriana Monaco** for being a constant guide and, above all, a source of inspiration.

I deeply thank **my mom, dad, and my sisters , Claudia and Manuela**, for always being the main pillars of all my achievements and successes, for enduring every moment of my stress, for celebrating every small milestone with me, for the love you show me every day.
I am grateful to my grandmother **Antonia** for always being with me and for consistently giving me the strength and serenity needed to complete this journey.
I also thank **Giuseppe** because, since you became part of our family, you have believed in me and encouraged me to give my best.

I sincerely thank my friends **Luca** and **Ilenia** for believing in me and being a support during the most challenging times, with comforting messages and long phone calls. I also thank **Riccardo** for being a guide, and I thank **Martina, Luca, Fabio, Leonardo, Vanessa, Flavio, Isabella, Andreas, Giuseppe, and Salvo**.
Thank you, my friends, for giving me these fantastic years in Turin, for contributing to the person I am today and for enriching my life.

Thank you all for always believing in me.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial Intelligence

API

Application programming interface

AWS

Amazon web services

CPU

Central processing unit

LB

Load Balancing policy

EE

Energy Efficient policy

EL-RL

Energy consumption and latency policy

RL

Reinforcement Learning

RAM

Random access memory

OS	Operating system
DQN	Deep Q-Network
UX	User Experience

Chapter 1

Introduction

Nowadays, one of the most precious resources for human beings is software. When talking about this realm, it is known that the ability to effectively managing continuously evolving applications and adapting to global changes without compromising, but rather enhancing, user experiences, highlights the need for increasingly sophisticated automation systems capable of autonomously managing applications. The success of an application is in fact closely linked to its management and subsequent UX.
Consequently, over the years, more and more applications have been leveraging cloud platforms for their distribution. This preference stems from the fact that the cloud comprises a suite of components, software and resources specifically designed for this purpose, ensuring reliability and enabling the efficient hosting and management of our systems.

Within this virtual world, it is crucial to manage all components instantly. Once again, software plays a pivotal role, especially in the form of orchestrators like Kubernetes [1], which automate the entire process of resource management. Kubernetes, in particular, is a platform for containerized applications in the cloud, representing a fundamental pillar in the world of distribution and utilization of applications.

One of the key aspects that have made Kubernetes a standard in the cloud world is its open-source nature. This fundamental characteristic, combined with the great flexibility and modularity of this project, allows for constant experimentation and improvement of the services offered in this field. For example, a very interesting case of possible improvements to its functionality consists of the intriguing union between the Cloud world and Artificial Intelligence, that is a powerful tool that is demonstrating all its potential in this century.

The object of this thesis is, therefore, the exploration of the potential of merging the Kubernetes orchestration software, in particular the scheduler component, with a DQN-type Reinforcement Learning model.
The reasons that led me to choose this fascinating theme as the main topic of my thesis are certainly associated with my growing interest in the world of AI and my curiosity to better understand those systems that, almost working in the shadows, improve users' lives without them being aware of it. Moreover, what truly fascinated me was the possibility of merging these two complex worlds through this work, thus giving me the opportunity to delve into these two spheres simultaneously.

1.0.1 Objective

The objective of this thesis is to compare the behavior of a system with Kubernetes containing a default scheduler versus our implementation of a scheduler plugin that utilizes Reinforcement Learning.With objectives similar to those presented in [2], we trained models to learn three main load management policies.
The result obtained is that one of the policies learned by the model, the policy *EL-RL*, has proved to be better than the default Kubernetes scheduler configuration. This work, therefore, serves as a basis for demonstrating, along with other works, the validity of using RL algorithms in container orchestration.

1.0.2 Methodology

The approach used to achieve our result involved initially defining a plugin inserted in the scoring phase for the Kubernetes scheduler. The Kubernetes scheduler is indeed the key component that determines on which cluster node a particular task should be started.

The system was then extended by leveraging Prometheus [3] to collect all the monitoring metrics needed during the model's training and testing phases. The metrics collected concern CPU, memory, and network values. However, a metric that is not directly exposed is latency, which in our solution is obtained by deploying an app on each node that, upon request, returns the latency between the node and the external user.

The other fundamental element is the DQN model. Derived from the cleanrl library [4] and subsequently modified, this DQN agent was implemented using a neural network that leverages the concept of DeepSet [5] and masks. This was a fundamental step within our system as the agent is capable of working with inputs of variable sizes and can work with values permutations. This aspect is crucial in the world of networks, given that a node could fail at any moment.

Once the system was defined, the next step was to define valid reward functions so that the model could learn a specific policy, along with the need to derive the various parameters to adapt the model to the problem. The model communicates its suggestions to a suggestion server that represents the meeting point between the plugin and the agent itself.Therefore, when a user submits a new scheduling request defining minimum requirements, the model retrieves the metrics by communicating with Prometheus and the various nodes, and sends the suggestion. This suggestion is then used by the scheduler to determine the node on which to start the task.
Upon completing this phase, testing of the implementation was carried out to compare the behavior of the systems with and without the use of our solution.

1.0.3 Structure

The main topics covered in this thesis are:

1. Related works: works related to what has been produced in this thesis
2. Background: a more in-depth overview of the systems used
3. System design: detailed architecture of the system and its functioning
4. Results: a summary collection of the results obtained
5. Conclusion: final considerations and possible future works

1.0.4 Contributions

The result obtained through hard work leads me to hope for a future in this field, which could, for example, concern one of the possible future works illustrated in the conclusions. This thesis, therefore, is a contribution to encourage the exploration of the use of AI algorithms in the orchestration of lightweight virtual machines.

The main contributions of this thesis in the end are:

- The definition of an EL-RL policy that enables training an RL agent to reduce energy consumption and latency between nodes and users when scheduling a new task in a Kubernetes-based system.
- Demonstrating that DQNs are powerful RL models suitable for Kubernetes orchestration, as they can bring significant improvements.

Chapter 2

Related Work

Kubernetes is an open-source platform for the orchestration of containerized applications that revolutionizes the management of modern software architectures by automating critical tasks such scalability, deployment, and management. One of its key architectural component is the scheduler, which is the responsible for allocating resources by determining the optimal cluster node on which to execute a new task, ensuring that the kubelet can perform it.

The main advantage of using Kubernetes lies on its flexibility, that provide to users the ability to adapt and configure the scheduler according their specific needs by means of default algorithms based on predefined policies. Additionally, it is also possible to create custom schedulers or integrate plugins to modify and customize the scheduling process in the cluster.

This chapter examines works related to modifications to the Kubernetes scheduler to improve the scheduling process.By examining various studies and initiatives, we uncover strategies and advancements geared towards optimizing workload placement, enhancing resource allocation, and improving overall cluster efficiency. These modifications play a crucial role in unlocking the full potential of Kubernetes, ensuring seamless orchestration of containerized applications in diverse and dynamic environments.

2.1 Custom scheduling

The concept of customizing the Kubernetes scheduler has been explored since as early as 2017. These customizations aim to address specific requirements that the default scheduler may not fully accommodate.

For example, in [6], the Kubernetes scheduler is modified to receive information from client applications to evaluate the best scheduling configurations. The motivation behind this is the inherent limitation of containers that can provide less isolation compared to vitrual machines. By integrating client-side information, the scheduler can make more informed decisions, mitigating resource contention issues.

Another contribution is described in [7], where NBWGuard is introduced. Here the proposed solution extends Kubernetes to treat network bandwidth as a schedulable resource„ integrating it into resource specifications alongside CPU and memory. Traditional Kubernetes scheduling primarily considers CPU and memory, which can lead to suboptimal performance for bandwidth-intensive applications. NBWGuard also supports the 3 kubernetes ’ QoS classes - Guaranteed, Burstable, and Best-effort- with respect to network bandwidth.

2.2 Latency aware scheduling

Another example is [8]. In this work the authors propose a latency-aware scheduler integrated into Kubernetes. In this case the scheduler is designed to guarantee that the latency between end-user devices and the replica placement is minimized. By prioritizing latency reduction, this customization enhances user experience and application responsiveness.

2.3 Reinforcement learning based scheduling

More recent works have explored the integration of machine learning techniques into Kubernetes scheduling.

The integration of RL enables the scheduler to dynamically adapt its decision-making process based on feedback from the environment, optimizing task placement and resource allocation in real-time. By leveraging RL, Kubernetes can enhance its scheduling capabilities, leading to improved efficiency and performance in diverse and dynamic computing environments.

This integration represents a significant advancement in Kubernetes scheduling, highlighting the potential of machine learning techniques to augment traditional scheduling algorithms and address the evolving needs of modern application deployments.

2.3.1 RLKube

In [2], the scheduler is extended by designing a Reinforcement Learning (RL)-based custom plugin that comes into action during the PreScore phase of scheduling, as it is shown in 2.2

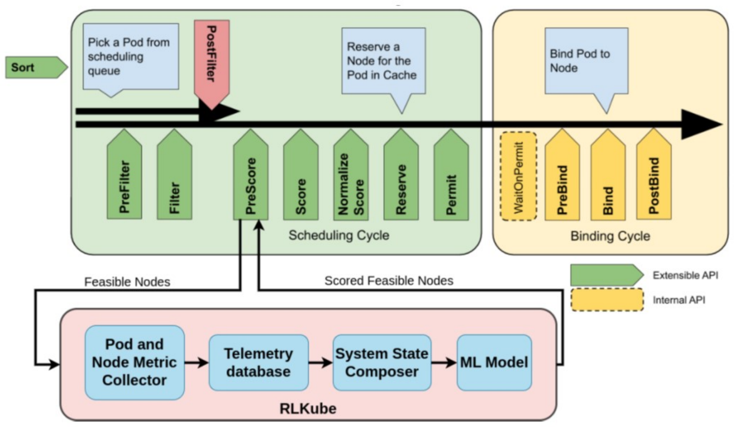


Figure 2.1: Kubernetes scheduling framework with additional RLKube Plugin (from [2])

In this solution, Prometheus is used to collect metrics on the state of the nodes, which are then forwarded to the RL model based on a Double Deep Q-Network. RLKube provides an evaluation in terms of node scores, which is then used in the final scheduling decision. The node with the highest score is selected for task execution. The objective is to optimize resource usage and improve the energy efficiency of the cluster by leveraging the advantages offered by reinforcement learning.

2.3.2 DRL-FORCH

In [9] another neural network-based solution is proposed. This study focuses optimize completing QoS requirements while working with fog nodes.

The orchestrator is implemented as Deep Set (DS) network and uses Deep reinforcement learning with invalid action masking to find an optimal trade-off between competing objectives.

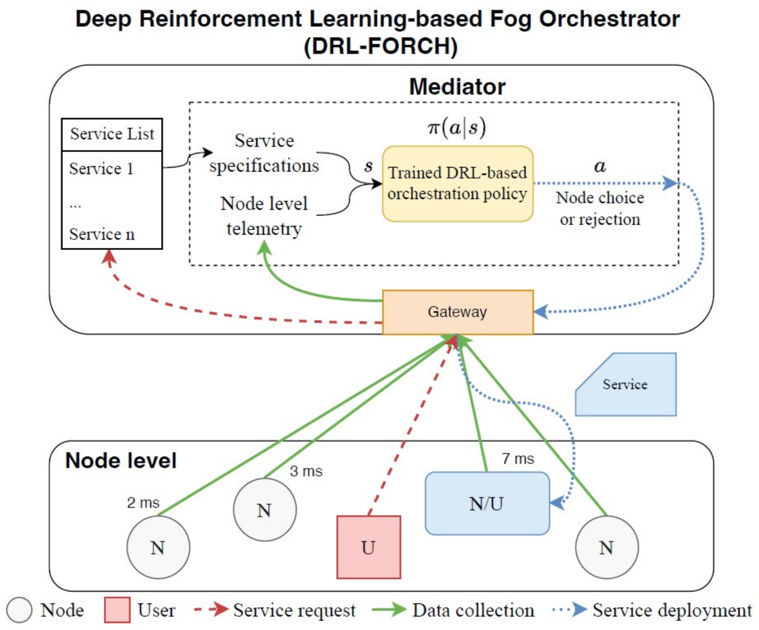


Figure 2.2: Service deployment decision making in DRL-FORCH (from [9])

Results show that the DS-based policy generalizes well to larger problem sizes, outperforming greedy heuristics and traditional MLP-based DRL. Additionally, the DS-based policy offers significantly faster inference times, enhancing scalability and enabling near real-time decision-making.

2.4 Energy consumption based scheduling

Another significant theme is the reduction of energy consumption in data centers, as demonstrated by AWS [10] and [11]. This has led to the proposal of many studies, such as those presented in [12, 13, 14].

In the study by [12], a new scheduling policy is introduced, aiming to migrate electricity consumption to regions with lower carbon intensity. Similarly, in [13], KEIDS is introduced as a controller to manage containers on edge-cloud nodes while considering carbon emissions.

In the work of [14], instances used by clients to deploy containers or execute computing tasks are considered. Heats is presented as a new task-oriented and energy-aware orchestrator for containerized applications targeted at heterogeneous clusters. It enables clients to trade performance for energy requirements by learning the performance and energy characteristics of physical hosts, monitoring task execution, and opportunistically migrating them across cluster nodes. Heats is implemented within Google Kubernetes and evaluation on synthetic traces indicates significant energy savings (up to 8.5%) with minimal impact on overall task execution time (at most 7%). Furthermore, Heats is released as open-source.

Chapter 3

Background

3.1 Kubernetes

Kubernetes [1] is an open-source, extensible, and portable platform for managing containers.

A container is a lightweight virtual machine that provides a less stringent isolation model compared to full virtual machines. It utilizes the underlying host's kernel, which allows the container image to be minimal, containing only the essential components of a specific operating system. This results in a much lighter and easy to use version of the OS.

By launching a simple command, Kubernetes creates automatically a virtualized cluster, which is a collection of nodes that will manage containers. Each node can host one or more pods, and the cluster ensures efficient distribution and management of workloads. Nodes can be physical machines or virtual machines, and they are responsible for running the containerized users's applications.

Key Components

- **Master Node**: This is the control plane of the Kubernetes cluster, responsible for managing the overall cluster. Among its various tasks, it hosts the *kubectl* client, which allows users to manage the cluster remotely. It includes components like the *scheduler* that has the role of assigning pods to nodes based on resource availability.
- **Worker Nodes**: These nodes are responsible for running the applications within containers.

For each cluster, there is at least one worker node, which runs the user's workload, and a master node or control plane that manages the workers and the cluster.

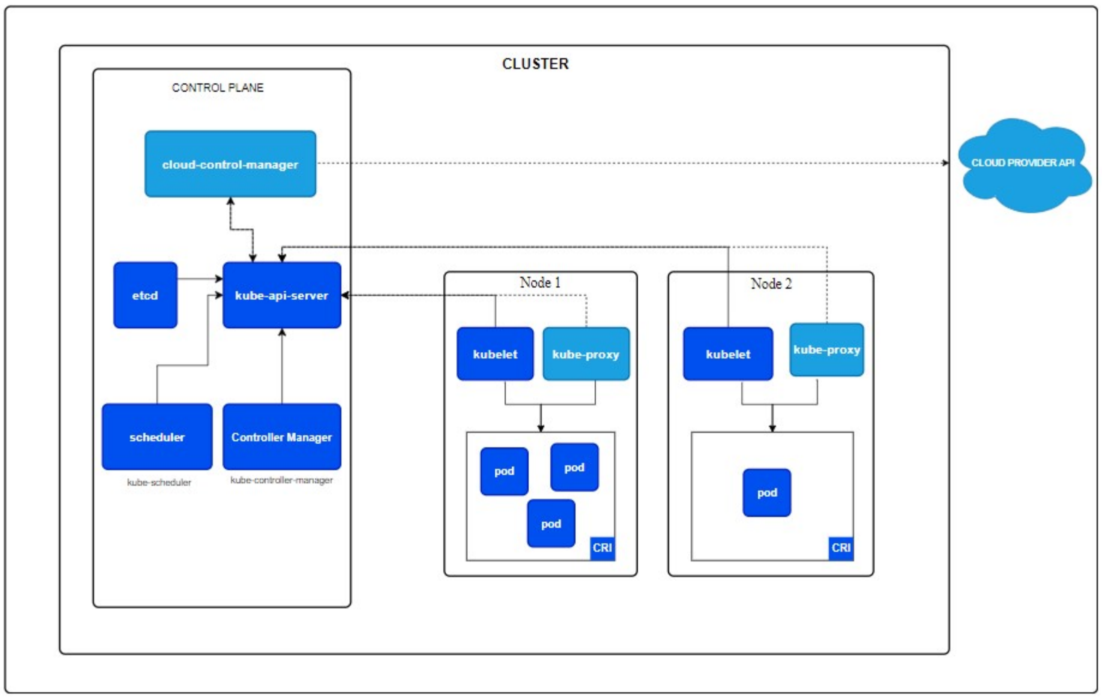


Figure 3.1: Kubernetes cluster architecture by [1]

3.2 Kubernetes Scheduling

A fundamental component for managing the Kubernetes cluster, running on the control plane, is the **scheduler**. This component observes newly created Pods and determines the best node on which to launch each new Pod. The default scheduler for Kubernetes is the **kube-scheduler**, which runs in the control plane. This component is designed to be **extensible**, allowing users to write their own scheduling components and use them.

Nodes that meet the scheduling requirements are called *feasible nodes*. The scheduler identifies these feasible nodes and then executes a set of functions to filter and score them, selecting the node with the highest score.

The node selection operation consists of two parts:

- **filtering** : In this phase, feasible nodes are filtered by executing a set of filtering functions, reducing their number so that the final selection will only consider a subset of nodes.

- **scoring** : In this phase, the remaining nodes are evaluated based on scores that can be attributed to nodes according to different elaborations or setting different plugins. The node that in the end has the higher score will be the chosen one.

3.2.1 Profiles

Filtering and scoring can be easily configured through **scheduling policies and scheduling profiles**. Specifically, profiles allow the configuration of existent plugins or custom plugins to implement one of the possible stages of filtering and scoring processes.

A possible kube-scheduler configuration that enables the custom plugin named "Network Traffic", as seen from [15], can be:

```yaml
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
clientConnection:
  kubeconfig: "/etc/kubernetes/scheduler.conf"
profiles:
- schedulerName: default-scheduler
  plugins:
    score:
      enabled:
      - name: NetworkTraffic
      disabled:
      - name: "*"
  pluginConfig:
  - name: NetworkTraffic
    args:
      prometheusAddress: "http://10.96.105.208:9090"
      networkInterface: "eth0"
      timeRangeInMinutes: 3
```

- `apiVersion: kubescheduler.config.k8s.io/v1` - Specifies the version of the Kubernetes Scheduler API to use.

- `kind: KubeSchedulerConfiguration` - Indicates that this configuration is for the Kubernetes scheduler.
- `clientConnection` - Contains settings for the scheduler's connection to the Kubernetes API server.
  - `kubeconfig: "/etc/kubernetes/scheduler.conf"` - Specifies the path to the kubeconfig file that the scheduler uses to connect to the API server.
- `profiles` - Defines scheduling profiles, which can be used to customize the scheduling process.
  - `schedulerName: default-scheduler` - The name of the scheduler profile.
  - `plugins` - Specifies plugins to be used during scheduling.
    - `score` - Plugins used for scoring nodes.
      - `enabled` - Lists plugins that are enabled for scoring.
      - `- name: NetworkTraffic` - The `NetworkTraffic` plugin is enabled for scoring.
      - `disabled` - Lists plugins that are disabled for scoring.
      - `- name: "*"` - Disables all other scoring plugins.
  - `pluginConfig` - Configuration for individual plugins.
    - `- name: NetworkTraffic` - Configuration for the `NetworkTraffic` plugin.
      - `args` - Arguments for the plugin.
      - `prometheusAddress: "http://10.96.105.208:9090"` - The address of the Prometheus server to query for network traffic data.
      - `networkInterface: "eth0"` - The network interface to monitor (e.g., `eth0`).
      - `timeRangeInMinutes: 3` - The time range in minutes for which to consider network traffic data.

So the Profile makes it possible to configure the different **stages** of scheduling in the kube-scheduler. Each stage is exposed in an **extension point**.

Plugins provide scheduling behaviors by implementing one or more of these extension points.

Scheduling is composed by the following list of stages exposed through the following extension points:

- **queueSort**: These plugins provide an ordering function that is used to sort pending Pods in the scheduling queue. Exactly one queue sort plugin may be enabled at a time.

- **preFilter**: These plugins are used to pre-process or check information about a Pod or the cluster before filtering. They can mark a pod as unschedulable.

- **filter**: These plugins are the equivalent of Predicates in a scheduling Policy and are used to filter out nodes that can not run the Pod. Filters are called in the configured order. A pod is marked as unschedulable if no nodes pass all the filters.

- **postFilter**: These plugins are called in their configured order when no feasible nodes were found for the pod. If any postFilter plugin marks the Pod schedulable, the remaining plugins are not called.

- **preScore**: This is an informational extension point that can be used for doing pre-scoring work.

- **score**: These plugins provide a score to each node that has passed the filtering phase. The scheduler will then select the node with the highest weighted scores sum.

- **reserve**: This is an informational extension point that notifies plugins when resources have been reserved for a given Pod. Plugins also implement an Unreserve call that gets called in the case of failure during or after Reserve.

- **permit**: These plugins can prevent or delay the binding of a Pod.

- **preBind**: These plugins perform any work required before a Pod is bound. bind: The plugins bind a Pod to a Node. bind plugins are called in order and once one has done the binding, the remaining plugins are skipped. At least one bind plugin is required.

- **postBind**: This is an informational extension point that is called after a Pod has been bound.
- **multiPoint**: This is a config-only field that allows plugins to be enabled or disabled for all of their applicable extension points simultaneously.

In this thesis the custom plugin extends the score stage, by adding the model suggestion in evaluating nodes scores.

For each extension point, it is also possible to disable specific default plugins or enable a specific one by editing the configuration file seen previously. For example:

```

apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
  - plugins:
      score:
        disabled:
        - name: PodTopologySpread
        enabled:
        - name: MyCustomPluginA
          weight: 2
        - name: MyCustomPluginB
          weight: 1
```

This special configuration file makes it easy to enable or disable the plugin so that is easier to compare the custom scheduler behaviour with the default one.

3.2.2 Scheduling pods with deployments

A Deployment is a kubernetes resource object used to deploy pods in a declarative way. This means that the user can define the requirements needed for the specific application and Kubernetes will manage the life cycle of pods autonomously.

The user describes a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. In the following case the deployment will ensure the scheduling of 3 replicas for and nginx app:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

- `apiVersion: apps/v1` - Specifies the API version for the Deployment resource. Here, it uses the apps/v1 API.
- `kind: Deployment` - Indicates that this resource is a Deployment.
- `metadata` - Contains metadata for the Deployment.
  - `name: nginx-deployment` - The name of the Deployment.
  - `labels: app: nginx` - Labels used to categorize the Deployment.
- `spec` - Defines the desired state of the Deployment.
  - `replicas: 3` - Specifies the number of pod replicas to maintain. Here, it is set to 3.
  - `selector` - Defines how the Deployment finds which pods to manage.
    - `matchLabels: app: nginx` - Matches pods with the label `app: nginx`.

- `template` - Describes the pods to be created.
  - `metadata` - Metadata for the pods.
    - `labels: app: nginx` - Labels for the pods.
  - `spec` - Specification for the containers in the pod.
    - `containers` - Defines the container specifications.
    - `name: nginx` - The name of the container.
    - `image: nginx:1.14.2` - The Docker image to use for the container, set to version 1.14.2 of Nginx.
    - `ports` - Specifies the ports exposed by the container.
    - `containerPort: 80` - Exposes port 80 on the container.

In Kubernetes, the management of various components is extremely simplified because changes to the system are declarative. The user defines the desired state and applies this specific configuration to the cluster. However, Kubernetes itself is responsible for achieving and maintaining the desired state.

3.3 Reinforcement Learning

Reinforcement learning [16] is a technique for understanding and automating **decision-making** and **goal-oriented learning**. It is used to discover new solutions and relies on neural networks that handle unstructured, unordered, and highly varied data. This type of learning appears to be the best method for making a machine creative in exploring new solutions.

Reinforcement learning models learn to make a series of judgments. Essentially, starting from an unpredictable and complex environment, the agent must learn on its own to achieve a goal. The environment is like a game, where the solution is found after many trials and errors. The AI receives rewards and penalties based on the outcomes of its choices. The objective is to maximize the total reward. The model designer sets the reward policy, or the rules of the game, which the model must maximize.

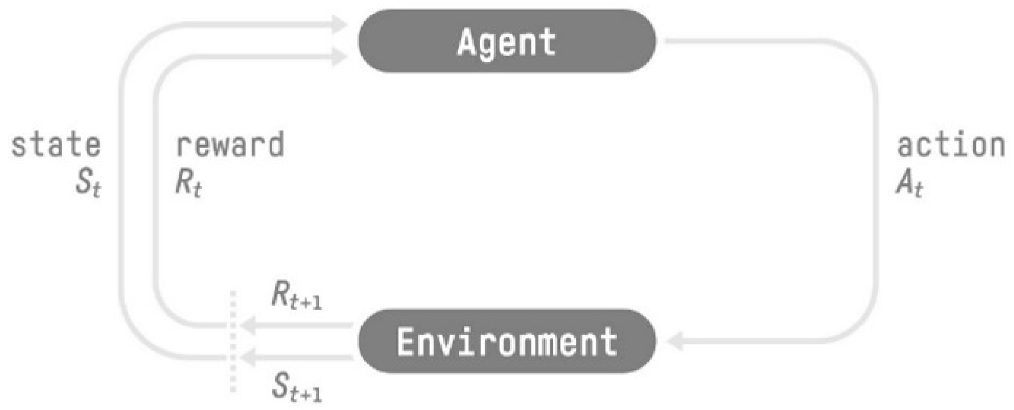


Figure 3.2: RL interaction between agent and environment from [16]

The most difficult aspect is setting up the simulation environment, which depends on the task at hand. Integrating the model into the real world is complex. Moreover, the only way to communicate with the network is through rewards and penalties.

Keywords

In the context of Reinforcement Learning (RL), **agents** are entities or computer programs that learn to take actions in an environment to maximize long-term **reward** or **evaluation**.
Agents are central in RL problems and play an active role in interacting with the environment and in the learning process. Here are some key concepts related to agents in Reinforcement Learning:

- **Goal** : The agent's task is to maximize the cumulative reward across various action steps in the environment before an episode ends.
  This is done to find the best solution among all possible solutions to the problem. Reinforcement learning (RL) is based on the reward hypothesis, which states that *all goals can be described as the maximization of the final outcome we want to achieve or as the maximization of the sum of partial outcomes.*

- **Policy**: An agent operates by following a *policy*, which is a strategy or probability map specifying which action to take in a specific state. The policy can be deterministic (e.g., action A in state S) or stochastic (e.g., probability of action A in state S).

- **Learning**: RL agents learn to improve their policy through interaction with the environment. They use various machine learning techniques to update their policy to maximize total rewards.

- **Exploration and Exploitation**: Agents must balance executing actions that have led to high rewards in the past (exploitation) with executing new actions to discover better actions (exploration).
  Finding the right balance between exploitation and exploration is a fundamental challenge in RL. In the exploration phase, the agent tries random actions to gather information about the environment, which is better than acting blindly.
  In the exploitation phase, we use this information to start maximizing the reward, thus acting with a more informed strategy. In general, there is a trade-off between exploration and exploitation, as one must determine how much to explore and when it is better to exploit the information gathered.

- **Evaluation and Control**: RL agents can engage in two main types of activities: "evaluation," which involves estimating the value of actions or policies, and "control," which involves optimizing policies to maximize reward.

- **State**:The state is a complete description of the world, meaning there is no hidden information that might force us to make blind decisions. For example, in the game of chess, we can see the entire board and evaluate all possible moves.

- **Observation** An observation is a partial information of the state. Some parts of the environment are hidden.

- **Applications**: RL agents are used in a wide range of applications, including robotics, games, resource optimization, autonomous management, machine learning, deep learning, and more.

3.4 RL approaches

How it is possible to build an agent that learns to choose an action aiming to achieve the maximum reward possible? The first step is to define a *policy*. A policy is a function that tells us which action to take given a certain state, thus defining **the agent's behavior** at any given moment. So, policy(state) = action.
This function is what the agent needs to learn and construct through training, and it is the function that maximizes the reward.

Thus, the idea is to define a reward function that assigns rewards each time the agent gets closer to understanding the policy.
The optimal policy can be found through two approaches:

- **Direct (policy-based)**: We teach the agent which action to take given the state. So in this case the agent learn the decision-making function directly, which maps a state to the best action to take either deterministically or based on the probability distribution of actions given a certain state, always evaluating our objective.
- **Indirect (value-based)**: We teach the agent to understand which state has the highest value and consequently to construct the policy function that brings us to the higher-value states in the environment, thus maximizing the reward. In the value-based method, instead of learning this function, we learn a value function that maps a state to a value associated with being in that state. The state's value corresponds to the expected return we anticipate obtaining if we choose to go to that state and continue choosing the highest-value states (thus following this policy).

Q-Learning

It is a value-based method that uses a temporal difference approach to train the action-value function.
The function we aim to find is a **Q-function**, which determines the value of being in a specific state and taking a specific action in that state. *The "Q" stands for quality, referring to the value of the action for that state.* The value of the state is the cumulative reward we expect the agent to obtain by following the policy.

Training means obtaining the optimal Q-table on which to build the optimal Q-function.

Initially, the Q-table is useless because we would have to make completely arbitrary decisions, but as the agent explores the environment, it will update the Q-table, giving us an increasingly better approximation of the optimal policy.

If the environment has a large number of states, managing the Q-table becomes complex. Therefore, we use **deep Q-learning algorithms**, where instead of using a Q-table, a neural network is used to approximate Q-values for each action based on the state.

Q-learning is a tabular method and is not scalable. For example, in Atari environments, a single frame consists of an image of 210x160 pixels, each varying from 0 to 255 possible values.

In this thesis the model is obtained by means of a deep Q-learning algorithm.

Deep Q-Learning

Deep Q-Learning (DQN) is one of the first algorithms that demonstrated the success of deep learning in reinforcement learning (RL). It is widely used in value-based control problems.

In deep RL, an agent learns to behave by interacting with the environment, taking actions that yield rewards (positive or negative). By repeating this process many times, the agent trains itself, generating a function that allows it to make decisions based on the future state of the environment.

The agent takes an action and consequently receives a reward and the next state (which can be the new timestamp frame) of the environment, on which to base the next decision. This cycle repeats continuously at each step as a sequence of four elements:

$S_0(state), A_0(action), R_1(reward), S_1(nextstate)$

Difference between Q-learning and DQN

The main difference between a Q-learning algorithm and a Deep Q-Network (DQN) algorithm is in how they maintain and update Q-values.

Q-learning uses a matrix to store Q-values for each state-action pair. This approach works well for environments with a limited number of states and actions but becomes impractical for more complex problems due to the exponential growth of the matrix size as the state and action spaces increase.

In contrast, *DQN uses a neural network to approximate the Q-values.* This neural network takes the state as input and outputs the Q-values for all possible actions in that state. By using a neural network, DQN can handle much larger and more complex state spaces, as the network can generalize and learn patterns in the data, making it possible to approximate the Q-values without explicitly storing them for each state-action pair.

This generalization capability allows DQN to apply Q-learning to problems that are infeasible with a simple matrix representation, such as those involving high-dimensional state spaces like images or large-scale environments. The neural network's ability to approximate Q-values through training on experiences also enables DQN to perform well in tasks where the state-action space is continuous or vast, opening up a broader range of applications compared to traditional Q-learning.

3.5 Custom Environment for RL Agent

The custom environment for a Reinforcement Learning (RL) agent is a simulation or interface that models the interaction with the external word. This environment encapsulates the state, actions, rewards, and transitions that define the problem space in which the RL agent operates.

Components of a Custom Environment

A typical custom environment consists of the following components:

1. **Observation Space:** Defines the state representation of the environment.

It includes all relevant variables and metrics that the RL agent can observe to make decisions. For example, in a Kubernetes cluster management context, the observation space may include metrics like CPU usage, memory availability, network traffic, and pod scheduling information.

2. **Action Space:** Specifies the set of actions that the RL agent can take in response to observations from the environment.
   In Kubernetes, actions could involve selecting a node for pod deployment, scaling resources, or adjusting scheduling policies.

3. **Reward Function:** Determines the immediate feedback provided to the RL agent based on its actions.
   The reward function is crucial as it guides the agent towards learning optimal behaviors. Rewards in Kubernetes environments may reflect performance metrics, cost efficiency, or system stability.

4. **Step Function:** Governs the interaction between the RL agent and the environment.
   It defines how the agent perceives observations, executes actions, receives rewards, and transitions to new states.
   In Kubernetes, the step function could involve querying Prometheus for node metrics, calculating rewards based on scheduling decisions, and updating the environment state, as it is done in this work.

5. **Termination Conditions:** Specifies criteria for terminating an episode within the RL environment. This ensures that the agent learns within a defined scope and can handle various scenarios gracefully.

Creating a Custom Environment

Creating a custom environment involves defining these components in a structured manner that aligns with the problem domain and the objectives of RL training. To create a custom environment it is needed to:

1. *Define Observation and Action Spaces*: Identify and formalize the state variables (observations) and permissible actions (action space) within the environment. This step requires a clear understanding of the system being modeled and the relevant metrics for decision-making.

2. *Implement Reward Function*: Design a reward function that quantifies the desirability of agent actions based on the environment's objectives. The reward function should incentivize behaviors that contribute positively to system performance or efficiency.

3. *Develop Step Function*: Implement the step function that orchestrates interactions between the RL agent and the environment. *This function should handle state transitions, action execution, reward computation, and termination conditions.*

4. *Integrate External APIs (Optional)*: If interacting with external systems or data sources (for example Prometheus for node metrics), integrate APIs to fetch relevant information and update the environment state dynamically.

5. *Validate and Test*: Validate the custom environment by running simulations and tests to ensure that the defined observations, actions, rewards, and transitions align with expected behavior and learning objectives.

By carefully crafting a custom environment, RL agents can effectively learn optimal strategies and contribute to enhanced automation and decision-making capabilities.

3.6 DeepSets

Machine learning algorithms typically can only handle fixed-dimensional data instances as input and output.
However, it is quite common to encounter problems that require processing inputs that can vary.

Recently, researchers have been trying to expand these algorithms so that they can successfully handle cases where inputs or outputs are permutation-invariant sets rather than fixed-dimensional vectors. Some studies, such as [5], specifically consider scenarios where both inputs and outputs provided to a model are sets.
It is in [5] where the implementation of DeepSets is introduced.

The DeepSets architecture utilizes invariant and equivariant transformations to handle data sets, maintaining invariance or equivariance with respect to permutations of input elements. This architecture effectively models data sets where the order of elements is irrelevant, but the overall structure of the set is significant.

Invariant Model

The DeepSets model, an invariant model, operates on the principles of permutation-invariant functions.
In this case the model's output should not change if the order of the input elements is changed. For example, if you have a set of numbers 1, 2, 3, the order doesn't matter, so 3, 1, 2 should give the same result.
The workflow is summarized as:

1. **Instance Transformation**: Each element of the set is transformed individually into a representation. Think of this as converting each number into a meaningful piece of information.

2. **Summation of Representations**: All these individual representations are added together. This summation step ensures that the model doesn't care about the order of elements.

3. **Output Transformation**: The sum of the representations is then passed through another network to produce the final output.

4. **Optional Conditioning**: If there's additional information (like context), it can be used to adjust the transformations.

Equivariant Model

The goal of the equivariant model is to design layers of neural networks that are equivariant with respect to permutations of elements in the input x. A function is equivariant if the permutation of input elements results in a permutation of output elements.

If you permute (rearrange) the input set, the output should be permuted in the same way. This is useful when the structure of the output should directly correspond to the structure of the input.

A neural network layer $f_\Theta(x)$ is equivariant with respect to permutations if and only if all off-diagonal elements of Θ are tied together and all diagonal elements are equal. So the equivariant model ensures that if you permute the input set, the output is permuted in the same way.

Implementation

The equivariant function is:

$$f(x) = \sigma(\lambda I x + \gamma \max(\text{pool}(x))1),$$

that is a special type of function where the output is structured similarly to the input if the input elements are rearranged. This equivariant layer can be further manipulated to achieve other variations, and it has been observed that in some applications, this variation performs better.

Composition of Equivariant Functions

Since the composition of equivariant functions with respect to permutations is also equivariant, it is possible to construct DeepSets by stacking such layers.

This enables the creation of neural networks that respect the invariance or equivariance with respect to permutations of inputs, making them suitable for tasks where the input structure is a set.

Independence from Input Dimensions

To make the previously introduced DeepSets independent even of variations in input dimensions, masks are introduced.

Specifically, when the model needs to determine the chosen action, the data provided in the state is masked using a dedicated function defined in the environment.

```
#Action mask fuction from environment
def action_masks(self):
valid_actions = np.zeros(self.state.__len__(), dtype=bool)
valid_actions[self.state != -1] = True
return valid_actions
```

In this context, the model obtains a mask that indicates which of the possible actions are valid, thereby dynamically adjusting the probabilities of choice. This mechanism allows the model to adapt to varying input sizes effectively, ensuring robust performance across different scenarios.

The use of masks ensures that the DeepSets framework can handle input data of varying dimensions seamlessly, maintaining its ability to process and make decisions based on relevant features within the data. This approach enhances the model's flexibility and applicability in real-world applications where input sizes may vary dynamically, as it appens for example in the case of our cluster.

Chapter 4

System Design

4.1 Architecture

The architecture of the solution proposed in this thesis consists of five fundamental communication units, implemented using Python or Golang.

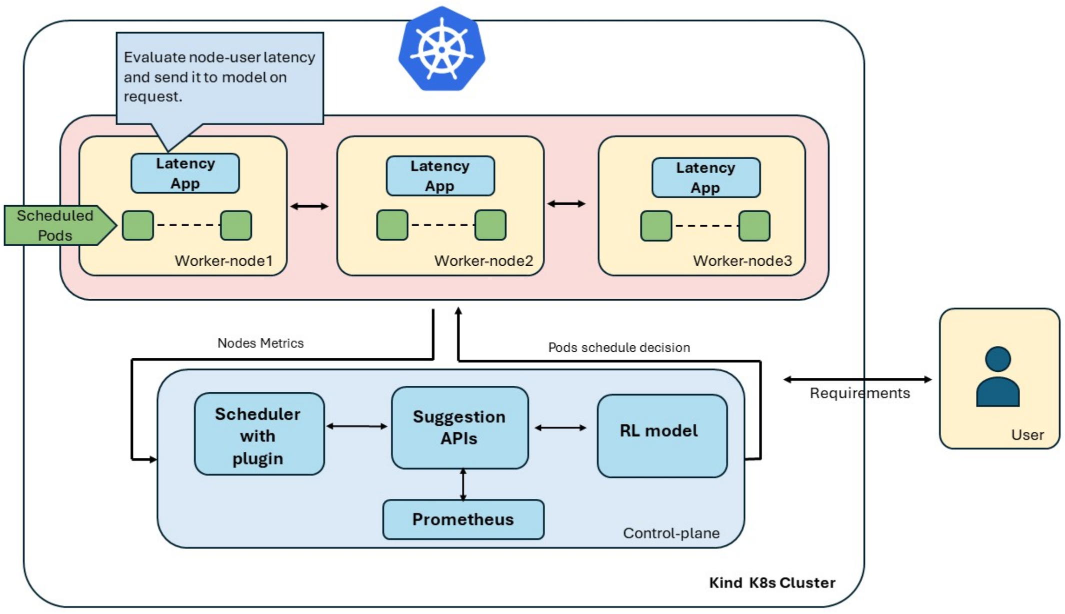


Figure 4.1: System architecture

As it is shown in figure 4.1, the Kind cluster was generated with 4 fundamental nodes. In detail, the main components developed are:

1 **Plugin**: This component plays a critical role during the scoring phase of the scheduling process. It requests scheduling suggestions from the RL model via exposed APIs.
By integrating these advanced decision-making capabilities directly into the scheduling workflow, the plugin ensures that node selection is based on dynamically computed metrics and real-time data.

2 **Suggestions Server**: The Suggestions Server exposes APIs for obtaining and setting scheduling recommendations. It acts as an intermediary between the scheduler and the RL model, facilitating seamless communication and ensuring that the scheduler can access up-to-date recommendations when making scheduling decisions.

3 **RL Model**: The Reinforcement Learning (RL) model is central to the intelligent decision-making process within the scheduler.
It utilizes node metrics, such as those fetched from Prometheus, to suggest optimal nodes for scheduling new pods. The model evaluates these metrics and provides a score for each node, indicating its suitability for handling additional tasks based on current network conditions and historical performance data.

4 **Prometheus**: This monitoring system is responsible for collecting and exposing various metrics from the nodes. These metrics include vital performance indicators such as CPU usage, memory consumption, and network metrics.
Prometheus serves as the data backbone, supplying the RL model with the necessary information to make informed and accurate scheduling decisions.

5 **Latency Application**: Deployed on each node, this application evaluates node-user latency upon request. It provides real-time data on the latency experienced by users interacting with different nodes. This information is crucial for determining the overall efficiency and responsiveness of the cluster, ensuring that user experience remains optimal.

These components work together to enhance scheduling efficiency and monitor system performance, leveraging metrics and real-time recommendations for effective resource management.

Thus, the user submits a new scheduling request describing the specifications regarding resource usage limits and latencies. This request reaches the scheduler, which requests a suggestion from the model via the suggestions APIs. The plugin then receives a score suggestion for the various nodes, with the node having the highest score being considered for task scheduling.
The choice to use a plugin allows for the easy enabling or disabling of RL functionalities simply by modifying a configuration file.

4.2 Scheduler Plugin

Kubernetes is renowned for its exceptional flexibility and extensibility, making it a preferred choice for various complex orchestration tasks.
During the scheduler design phase, we carefully evaluated our options to determine the best approach for our specific needs. We considered two main possibilities:

- Rewrite a new scheduler from scratch and configure its image within the Kubernetes kind cluster.
  This approach would involve developing a completely custom scheduler tailored to our requirements, providing us with full control over its behavior and features.

- Create a plugin that could be seamlessly integrated into any existing scheduler.
  This option focuses on developing a modular extension that can enhance the functionality of the default scheduler or any other scheduler without the need to replace it entirely.

After thorough analysis and consideration, we opted for the second approach.
Creating a custom plugin offered several advantages: it allowed us to leverage the robust and well-tested default scheduler while introducing additional capabilities specific to our use case. This approach minimized the complexity associated with maintaining a completely custom scheduler and ensured compatibility with future updates to the Kubernetes ecosystem.

As a result, *the final scheduler in our Kubernetes environment is the default scheduler augmented with our custom plugin.* This solution strikes a balance

between leveraging existing, stable technologies and introducing the flexibility needed to meet our unique scheduling requirements. By opting for a plugin-based approach, we achieved an elegant and maintainable solution that integrates seamlessly into the Kubernetes architecture.
The plugin developed for the scheduler is implemented in Golang, utilizing [17] as the foundation for the modifications made, which in turn is introduced in [15].

To thoroughly study and evaluate its functionality, all default plugins have been disabled. This leaves only the custom plugin active, ensuring that its effects on the scheduling process can be observed in isolation without interference from other default scheduling behaviors.

The plugin specifically intervenes when a new task is scheduled during the scoring phase of various nodes (4.2). During this phase, each node is assigned a score that reflects its suitability for executing the task. This suitability is assessed based on evaluations performed within the plugin, which takes into account RL model suggestions to optimize node selection.

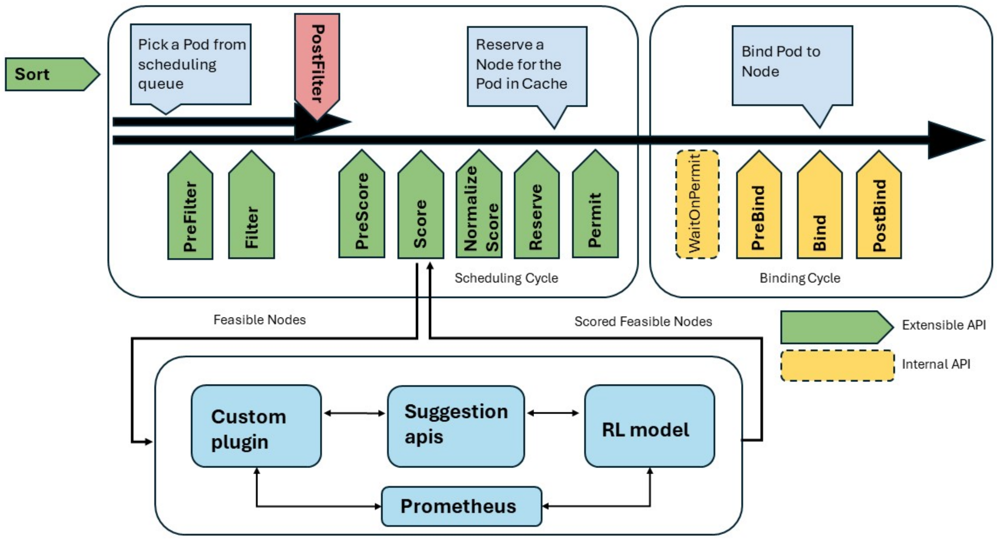


Figure 4.2: Scheduler plugin Architecture (scoring phase)

The scoring mechanism is driven by a Reinforcement Learning (RL) agent, which operates as a process within the scheduler. The RL agent evaluates relevant metrics to assign scores to each node. These scores are then used to determine the optimal node for task execution, aiming to enhance overall performance and efficiency of the Kubernetes cluster optimizing resource usage.

The integration of the RL agent within the scheduler is a key innovation, as it brings adaptive learning capabilities to the scheduling process. By continuously learning from the environment and task execution outcomes, the RL agent can dynamically adjust its scoring criteria, leading to progressively better scheduling decisions over time.

4.3 RL Model

The Reinforcement Learning (RL) agent utilized in this architecture is a Deep Q-Network (**DQN**) agent from the library **cleanrl** [4].
Significant customizations have been applied to tailor the agent's operations within a custom environment designed to simulate the Kind cluster's conditions accurately. This custom environment is crucial as it provides all the necessary information to the agent for both training and testing phases, ensuring that the learned policies are directly applicable to the real cluster scenario.

One of the major modifications to the standard DQN agent involves the incorporation of **DeepSets** into the neural network structure, as it is done in [9]. DeepSets are a powerful neural network architecture that allows the model to handle input data permutations correctly and adapt to changes in input dimensions. This adaptability is particularly useful in realistic scenarios where a node may not be temporarily active, leading to a dynamic and variable number of nodes in the environment.

The modified DQN-RL agent is modified as follows:

```
class EquivariantLayer(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.Gamma = nn.Linear(in_channels, out_channels,
    bias=False)
```

```python
        self.Lambda = nn.Linear(in_channels, out_channels, bias=False)

    def forward(self, x: torch.Tensor):
        #pdb.set_trace()
        gamma_x = self.Gamma(x)
        xm, _ = torch.max(gamma_x, dim=1, keepdim=False)
        return self.Lambda(x) - self.Gamma(xm.unsqueeze(1).expand_as(x))




class QNetwork(nn.Module):
    def __init__(self, env):
        super().__init__()
        self.final_layer = nn.Linear(env.single_action_space.n, 3)

        self.network = nn.Sequential(
            EquivariantLayer( envs.observation_space.shape[1], 120),
            nn.ReLU(),
            EquivariantLayer(120, 84),
            nn.ReLU(),
            EquivariantLayer(84, env.single_action_space.n),
        )

    def forward(self, x):
        x = self.network(x)
        return self.final_layer(x)

    def get_value(self, x: torch.Tensor):
        return self.critic(x)

    def get_action(self, x: torch.Tensor, masks: Optional[torch.Tensor] = None, deterministic: bool = True):
        logits = self.network(x)
        if masks is not None:
            HUGE_NEG = torch.tensor(-1e8, dtype=logits.dtype)
            if masks.dim() == 1:
                masks = masks.unsqueeze(0)
            logits = torch.where(masks, logits, HUGE_NEG)
        # discrete probability distribution over a set of actions. The logits provide the unnormalized log probabilities for each action.
        dist = Categorical(logits=logits)
        # if deterministic is True, return the mode of the Categorical distribution (highest probability, selecting the action with the highest logit value)
        if deterministic:
            return dist.mode
        # if deterministic is False, return a random sample from the Categorical distribution.
        return dist.sample()

    def get_feature_size(self):
        return self.feature_size

    def predict(self, obs: npt.NDArray, masks: Optional[npt.NDArray] = None) -> npt.NDArray:
        with torch.no_grad():
            action = self.get_action(
                torch.as_tensor(obs, dtype=torch.float32), torch.as_tensor(masks, dtype=torch.bool), deterministic=True
            ).numpy()
        return action
```

To implement DeepSets, the neural network's standard Linear Layers are substituted with **Equivariant Layers**. These layers are designed to maintain the functionality of DeepSets, ensuring that the network can process sets of nodes as inputs rather than fixed-size vectors. This change enhances the model's ability to generalize across different cluster states and configurations, making it robust to variations in the number of active nodes.

Additionally, two new functions, **'predict'** and **'get action'**, have been integrated into the neural network. These functions are critical during the testing phase:
The **'predict'** function is used to extract the value with the highest probability from the distribution of values obtained during training. This function

helps in making deterministic decisions based on the learned policy.
The **’get action’** function assists in extracting the best possible action to take in the current state based on the trained model’s predictions. It ensures that the agent selects actions that maximize the expected reward, adhering to the policy learned during training.

During the training phase, however, the action selection process involves an element of randomness. The agent chooses actions randomly from the set of valid actions provided by the environment. This approach, known as exploration, is crucial for the agent to learn effectively as it helps in discovering new strategies and refining the policy over time. The environment supplies a valid actions mask, ensuring that only permissible actions are considered during this exploratory phase.

Models were trained to learn a specific policy by varying the reward function inserted in the custom environment. The reward function is a critical component of the RL framework, guiding the agent’s learning process by providing feedback on the quality of actions taken.
By adjusting the reward function, different aspects of scheduling performance, such as efficiency, latency, and resource utilization, can be emphasized, leading to a policy that aligns with the desired objectives.

4.3.1 Custom environment

To enable the RL agent to work effectively on the cluster, a custom environment was created and registered among the possible environments in the Gymnasium library [18]. Gymnasium, a popular toolkit for developing and comparing reinforcement learning algorithms, provides a standardized interface for environments, making it an ideal framework for our custom implementation.

Key Features of the Custom Environment

- **Action Space**: The custom environment defines a discrete action space. Each action corresponds to the selection of a specific node in the cluster where a new pod should be scheduled.
- **Observation Space**: The observation space includes all relevant information about the nodes in the cluster, such as CPU usage, memory

consumption, network latency, and other performance metrics.

Table 4.1: Action and Observation Spaces

Action Space	**Observation Space**
Discrete actions (nodes)	CPU, memory consumption, network latency, etc.

Step Function

The step function encapsulates all the steps required when the agent performs an action in the environment. This function is called each time the agent takes an action, and it updates the environment's state accordingly. Specifically, it involves:

– Action execution: The model selects a node to schedule a new pod.

– State update: The environment updates its state based on the action taken.

– Reward evaluation: The environment evaluates the outcome of the action and assigns a reward based on the defined policy.

Reward Function

The reward function is critical for guiding the agent's learning process. It provides feedback on the quality of the actions taken, influencing the agent's decision-making process.
The design of the reward function varies based on the specific policy objectives. For example, it may prioritize efficiency, latency reduction, or other performance metrics relevant to cluster management.

Environment implementation Details

The custom environment is integrated and registered with the Gymnasium library, ensuring compatibility and usability within the RL framework.
By defining a discrete action space and a comprehensive observation space, the environment enables the agent to make informed decisions and optimize cluster performance through intelligent scheduling strategies.

The custom environment plays a crucial role in facilitating effective RL-based scheduling on Kubernetes clusters. It provides the necessary framework for the agent to learn and adapt scheduling policies based on real-time cluster conditions, enhancing resource utilization and performance efficiency.

4.4 RL model policies

Energy efficient (EE) policy

According to this policy the goal for the model is to **learn to choose node with highest resource utilization** (CPU, memory, disk, or number of scheduled pods) to optimize energy consumption.

At each step the custom environment provides an observation of the cluster and in this case the provided observation is the number of scheduled pods per node that can be switched with a resource between CPU, memory, disk. So the model will choose a possible action selecting among available nodes the one that will get the maximum score.

For this policy the reward function simply is the ratio of unused machines to total machines based on resource utilization. In particular, at each step the reward is obtained as follows:

```
# EE REWARD
unused_nodes = sum(1 for value in self.state if value < 1)
reward = int( 0 if terminated else (unused_nodes/(self.
    num_of_nodes -1))*10 )
self.state=get_nodes_state(self.policy , self.worker_ips)
```

The episode is terminated when a node different from the one to select is chosen.

Load Balancing (LB) policy

According to this second policy the goal for the model is to **learn to choose node with lowest resource utilization** (CPU, memory, disk, or number of scheduled pods) to increase throughput.

At each step the custom environment provides an observation of the cluster and in this case the provided observation is the number of scheduled pods per node that can be switched with a resource between CPU, memory, disk. So the model will choose a possible action selecting among available nodes the one that will get the maximum score.

For this policy the reward function simply is a constant reward upon successful pod scheduling. In particular, at each step the reward is obtained as follows:

```
terminated = action.__eq__(node_to_not_select)
almost_terminated = action.__ne__(node_to_select) and action
    .__ne__(node_to_not_select)

# LB REWARD
reward = 10 if action.__eq__(node_to_select) else (0 if
    almost_terminated else -1)
self.state=get_nodes_state(self.policy, self.worker_ips)
```

In this case, the episode is terminated if the selected node is the worst (the most loaded), while the episode is "almost terminated" if the second most loaded node is selected.

Energy and Latency Reinforcement Learning (EL-RL) policy

In the last case the goal for the model is to **learn to select the node with lower node-user latency and highest number of scheduled pods** to optimize both energy consumption and latency.

At each step the custom environment provides an observation of the cluster and in this case the provided observation is the number of scheduled pods, memory and CPU usage, node-user latency per node. So the model will choose a possible action selecting among available nodes the one that will get the maximum score.

For this policy the reward function is the combination of rewards to minimize used nodes, host pods on minimal nodes, and choose node with best latency (values in range [0,1]) In particular, at each step the reward is obtained as follows:

```
# EL-RL REWARD
r1 = (self.num_of_nodes - on_nodes)/ self.num_of_nodes # reward member to minimize the number of used nodes
tot_pods = np.sum(scheduled_pods)
r2 = tot_pods/on_nodes # reward member to host pods on min number of nodes [0, 1]
r2_min = 0  # no pods scheduled
r2_max = 50  # all pods scheduled on the single on_node

if on_nodes > 0:
    r2 = (r2 - r2_min) / (r2_max - r2_min)
else:
    r2 = 0

latency_node_user = float(getLatency(action_node_ip))

r3 = 0
if latencyHardConstraint!= -1 and latencySoftConstraint!= -1:
    if latency_node_user > latencyHardConstraint:
        r3 = 0
        terminated = True
    elif (latencySoftConstraint< latency_node_user) and (latency_node_user <= latencyHardConstraint):
        r3 = 0.5
        terminated = False
    elif latency_node_user <= latencySoftConstraint:
        r3 = 1
        terminated = False
else:
    r3 = 0
    terminated = True

reward = r1 + r2 + r3
```

Here the episode is terminated when the latency requirement is not satisfied.

In this case, the goal is also to satisfy users' latency requirements. In particular, the user can define two constraints:

- **LatencyHardConstraint**: A requirement of latency that must be strictly satisfied.
- **LatencySoftConstraint**: A requirement of latency that does not need to be strictly satisfied but allows for a higher reward as it is the better choice.

To evaluate node-user latency, each node runs an app that, upon request, measures the latency between the node and the user. Therefore, the observation provided to the model will also contain all the latencies for all nodes, so that the agent's decision can also take latency into account.

In particular, as can be seen from the EL-RL reward code, the component r_3 provides the part of the reward related to latency, giving 3 possible values:

$$r_3 = \begin{cases} 0 & \text{if latency\_node\_user} > \text{HardConstraint} \\ 0.5 & \text{if SoftConstraint} < \text{latency\_node\_user} \leq \text{HardConstraint} \\ 1 & \text{if latency\_node\_user} \leq \text{SoftConstraint} \end{cases} \tag{4.1}$$

The components r_1 and r_2 will ensure that the maximum number of pods are scheduled on the minimum number of nodes. In particular, the goal is to select the most loaded node.

In general, therefore, the user can define specific requirements they wish to meet for running their application, and these requirements are taken into consideration during the reward or penalty phase for the agent, which is rewarded when all requirements are satisfied.

4.5 Suggestions Server

The suggestion server written in Python plays a pivotal role within the system architecture by exposing APIs that facilitate seamless communication between the scheduling plugin and the machine learning model. These APIs serve as the primary interface through which the plugin interacts with the model, exchanging scheduling recommendations and receiving real-time updates.

Additionally, the suggestion server provides APIs specifically designed to retrieve critical node metrics from the Prometheus monitoring system. This integration is essential for obtaining detailed insights into the current state of nodes within the Kubernetes cluster. By configuring Prometheus with Node Exporter, the system ensures comprehensive monitoring capabilities, capturing metrics such as CPU utilization, memory availability, disk usage, network traffic, and other performance indicators.

4.6 Prometheus

Prometheus [3] is a service monitoring system , renowned for its scalability and robustness in cloud-native environments, that employs a scraping-based architecture to gather metrics directly from cluster nodes and services. Prometheus can deliver accurate and timely data, essential for informed decision-making and efficient resource management.

The combination of the suggestion server's APIs and Prometheus' monitoring capabilities forms a robust foundation for enhancing cluster performance and reliability. By leveraging these technologies, the system not only optimizes resource allocation and workload distribution but also empowers dynamic adjustments based on real-time operational insights.

Specifically, in this solution, Prometheus has been configured using the **Node Exporter**. The Node Exporter exposes a wide variety of hardware- and kernel-related metrics such as CPU usage, memory utilization, disk usage, and network statistics from Linux and UNIX hosts.
By configuring Prometheus with the Node Exporter, this solution enables the collection of detailed performance metrics from each node in the Kubernetes cluster.
The main queries used in training are:

CPU

```
100 - (avg(irate(node_cpu_seconds_total{mode="idle",
    instance="{internal_node_ip}:9100"}[1m])) * 100)
```

This query calculates the CPU usage percentage by subtracting the average idle time from 100%.

MEMORY

```
node_memory_MemAvailable_bytes{instance="{internal_ip}:9100"
    }/10^9
```

This Prometheus query returns the amount of available memory in gigabytes on a specific node.

DISK

```
100 - (node_filesystem_free_bytes{mountpoint="/run",instance
    ="{internal_ip}:9100"} / node_filesystem_size_bytes{
    mountpoint="/run",instance="{internal_ip}:9100"} * 100)
```

This query evaluates the percentage of free space on the specific filesystem.

NETWORK TRAFFIC

```
rate(node_network_receive_bytes_total{instance="{internal_ip
    }:9100", device="eth0"}[5m])
```

This query calculates the rate of byte reception on the "eth0" interface over the last 5 minutes.

NETWORK CONNECTIONS

```
node_netstat_Tcp_CurrEstab{instance="{internal_ip}:9100"}
```

This Prometheus query retrieves the current number of established TCP connections on a specific node.

PODS NUMBER

```
count(kube_pod_info{{node="{node_name}", namespace="default"
    }}) or vector(0)
```

This Prometheus query retrieves the current number of pods scheduled in default namespace for each node.

The decision to integrate Prometheus into the cluster management system was driven by several key considerations related to resource monitoring and performance management. Prometheus was chosen for its robustness, flexibility, and ability to provide detailed real-time data on cluster nodes.

Prometheus is an open-source monitoring and alerting system originally designed to monitor cloud-native infrastructures like Kubernetes. Its architecture model based on scraping allows it to collect metrics directly from running services, providing an accurate and detailed view of system performance.

Node Metrics Monitoring

Prometheus integrates the Node Exporter to gather essential metrics such as CPU usage, available memory, disk space, and network traffic on each cluster node.
Many other metrics can be collected on the cluster, as prometheus can be configured in different ways.

Support for Scheduling Decisions

Metrics collected by Prometheus are used to train machine learning models, such as Deep Q-Network (DQN) agents, to recommend the most suitable nodes for scheduling new pods. For example, the model can leverage CPU and memory usage information to balance workload across nodes and enhance overall system efficiency.

Scalability and Flexibility

Due to its scalable architecture and extensive library support, Prometheus can be easily adapted and extended to meet specific monitoring and management needs of Kubernetes clusters. Its ability to handle large volumes of data and integrate with other automation tools makes it an ideal choice for dynamic and evolving environments like Kubernetes clusters.

Chapter 5

Results

Testing

Once the system is launched and configured, the testing phase is executed through various scheduling steps for the different policies. Specifically, each policy corresponds to a model that learns a specific behavior.

- **Scheduling Steps**: The system employs a series of scheduling steps to ensure that each policy is tested thoroughly. This involves deploying the policies under different conditions and monitoring their performance.

- **Policy**: Each policy implemented in the system is associated with a trained model. These models are designed to learn and adapt to specific behaviors based on predefined criteria. During the testing phase, the behavior of each model is evaluated to ensure it meets the expected outcomes.

- **Monitoring and Evaluation**: Throughout the testing phase, continuous monitoring and evaluation are conducted to assess the performance of each policy. This involves collecting data, analyzing results, and making adjustments as needed to improve the overall system performance.

By executing these testing steps, the system ensures that each policy functions correctly and efficiently within the Kubernetes Kind environment, providing valuable insights and validation for future deployments.

In particular, the testing phase consists of scheduling a certain number of deployments and deleting them, repeating the operation cyclically.

Versions

The implemented system is tested in a Kubernetes Kind environment with the following versions of the installed programs:

Program	**Version**
Go	go 1.21.4
Docker	Docker version 20.10.23, build 7155243
Ubuntu	Ubuntu 20.04.6 LTS
Kind	kindest/node:v1.27.3
Windows	11 Home 23H2
Python	Python 3.8.10

Table 5.1: Table of Versions

Deployment

For each pod, the specifications are defined in the deployment file, where the LatencyHardConstraint and LatencySoftConstraint are also specified.

In particular values are:

- LatencyHardConstraint: 30
- LatencySoftConstraint: 20

The deployment file that is cyclically created requires the scheduling of a pod of type nginx:latest.

```

    apiVersion: apps/v1
    kind: Deployment
    metadata:
      name: "nginx-deployment-25"
      labels:
        app: nginx
      annotations:
        latencySoftConstraint: "20"
        latencyHardConstraint: "30"
    spec:
```

```
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
      annotations:
        latencySoftConstraint: "20"
        latencyHardConstraint: "30"
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        resources:
          requests:
            memory: "64Mi"    # 64MB of memory
            cpu: "250m"       # (0.25 CPU)
```

This file is applied to the cluster using two different testing algorithms, which operate in two distinct ways:

- In the first case (**test1**), all pods are created initially and then subsequently deleted to repeat the scheduling cycle.
- In the second case (**test2**), pods are created continuously. Once the number of pods reaches approximately 30, the first pod in the list of created pods is deleted.

Test1

Algorithm 1 Kubernetes Deployment Management 1

```
for 500 iterations do
    for 30 iterations do
        generate deployment name
        call replace_deployment_name with deployment name
        create deployment in Kubernetes
        wait for 3 seconds
    end for
    wait for 20 seconds
    for 30 iterations do
        generate deployment name
        delete deployment in Kubernetes
    end for
    print "All deployments deleted."
    wait for 5 seconds
end for
```

Test2

Algorithm 2 Kubernetes Deployment Management 2

```
1: for 30 iterations do
2:     generate deployment name
3:     call replace_deployment_name with deployment name
4:     create deployment in Kubernetes
5:     Add $deployment\_name$ to $pod\_names$
6:     Wait for 2 seconds
7: end for
8: while true do
9:     $old\_deployment \leftarrow$ first element of $pod\_names$
10:    delete deployment $old\_deployment$
11:    Wait for 3 seconds
12:    generate deployment name
13:    create deployment in Kubernetes
14:    Remove first element from $pod\_names$
15:    Add $new\_deployment$ to $pod\_names$
16:    $next\_index \leftarrow next\_index + 1$
17:    Wait for 1 seconds
18: end while
```

5.1 Energy efficient (EE) policy

Test 1

The first test consist in scheduling a number of 30 pods and to delete all after 10 seconds. This scheduling is repeted ciclically. In this case, the result obtained in testing as it is shown in figure 5.1 is that the model will focus all the scheduling on one node that is the most loaded, whithout considering any latency.

In this case for example Node 0 is the one with the worst latency, but is chosen since it is the most loaded.

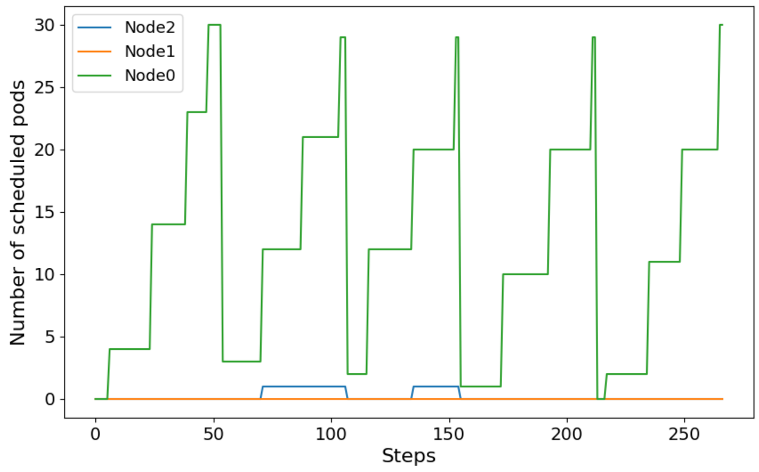


Figure 5.1: Energy efficient policy: pods distribution on test 1

Test 2

The second test consist in scheduling a number of pods each 2 seconds and at 30 pods scheduled the oldest scheduled pod is deleted while the system schedule other pods each 3 seconds. In this case, the result obtained in testing as it is shown in figure 5.2 is again that the model will focus all the scheduling on one node that is the most loaded, whithout considering any latency.

In this case the selected node is Node2, that still is not the best choice for latency.

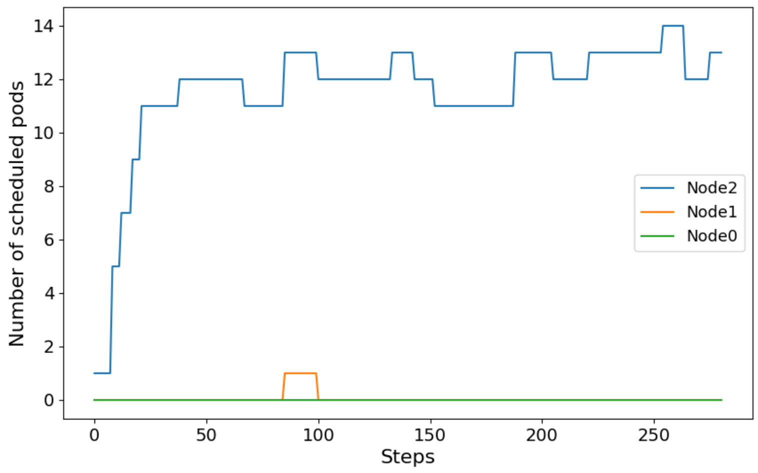


Figure 5.2: Energy efficient policy: pods distribution on test 2

5.2 Load balancing (LB) policy

Test 1

With this policy the goal is to learn to schedule pods between nodes so that the load is distributed.

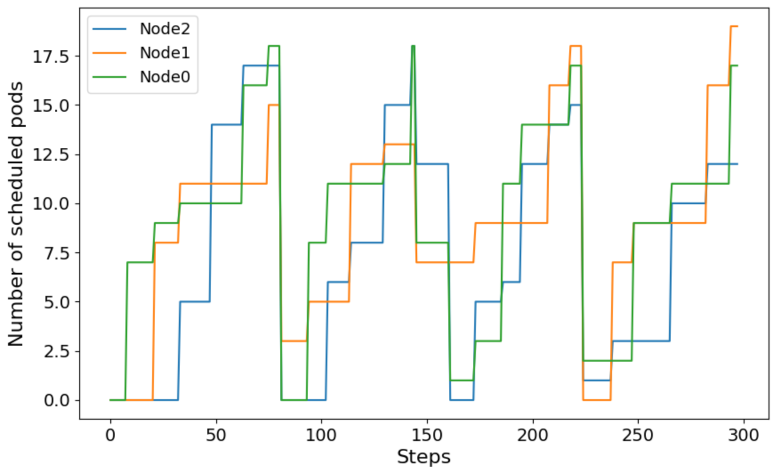


Figure 5.3: Load balancing policy: pods distribution on test 1

As it can be seen from the figure 5.3, in the first case of testing the desired result is obtained.

Test 2

In the second test case, as it is shown in figure 5.4 the load distribution is even more apparent as, at each step, the least loaded node is selected, even when old pods start to be removed.

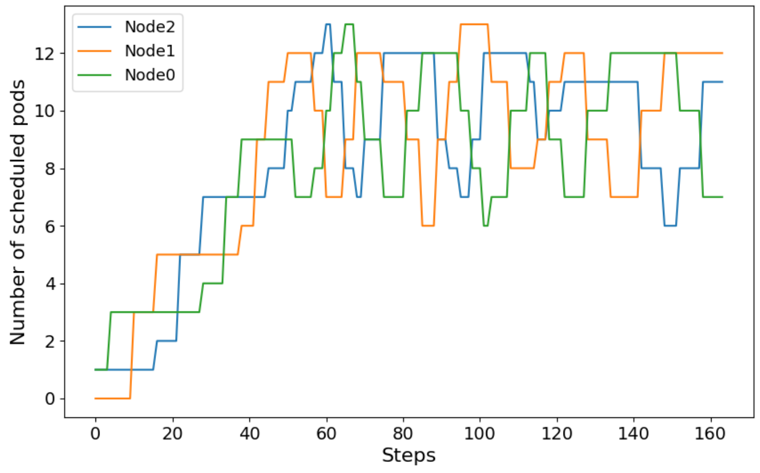


Figure 5.4: Load balancing policy: pods distribution on test 2

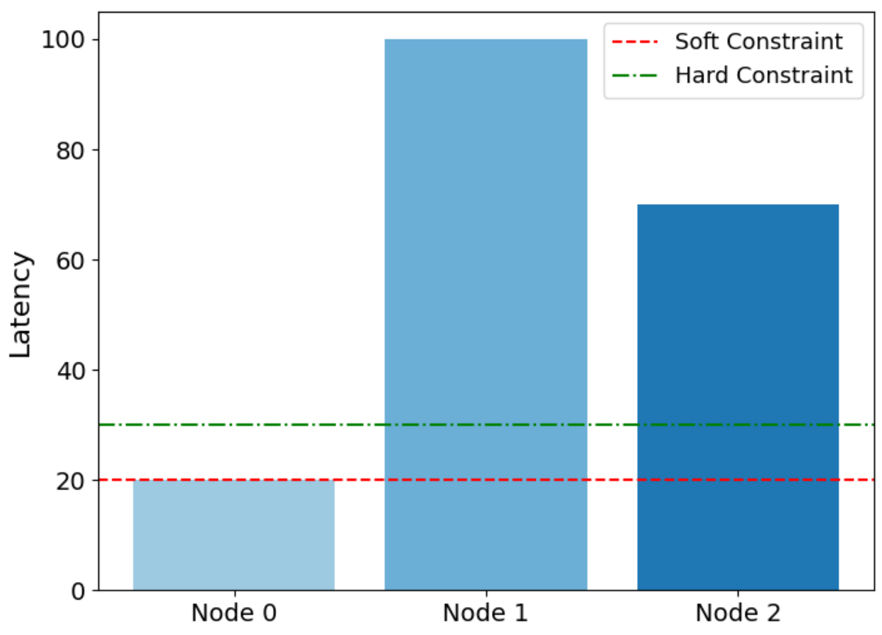
5.3 Latency and energy efficiency policy

test1

In this scenario, the training involves systematically varying latencies between nodes and users by introducing different latency values. This approach simulates significant fluctuations in latency during the training process.

In the first case node0 has lower latency since step 100, and it satisfies both soft and hard latency constaints. The latencies are varied to verify the model's learning.

The following images show the agent's choices in some of the possible latency scenarios.



Node0 with lower latency

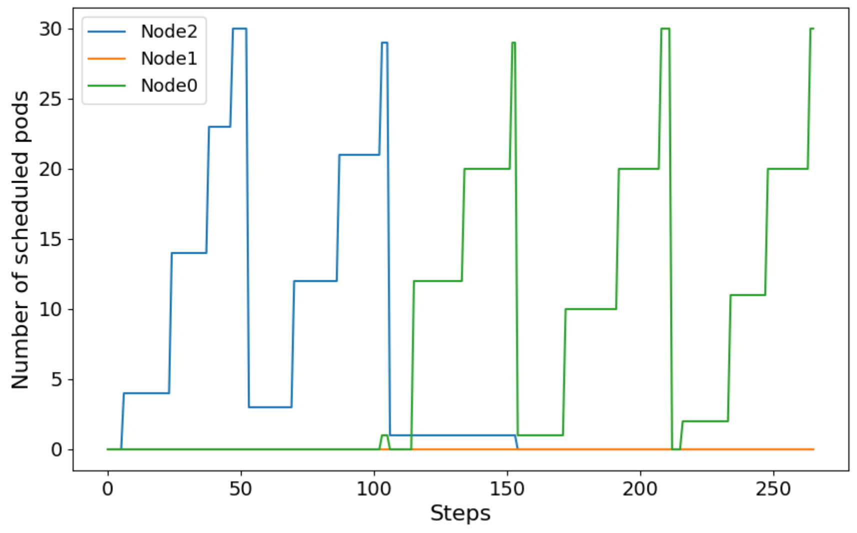
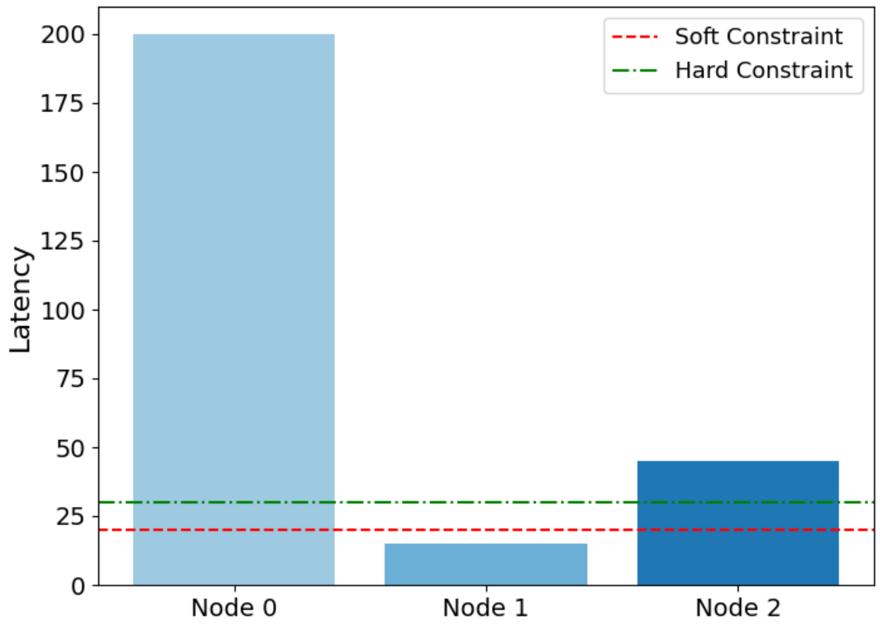


Figure 5.5: EL-RL Pods scheduling on node0 (test1)



Node1 with lower latency

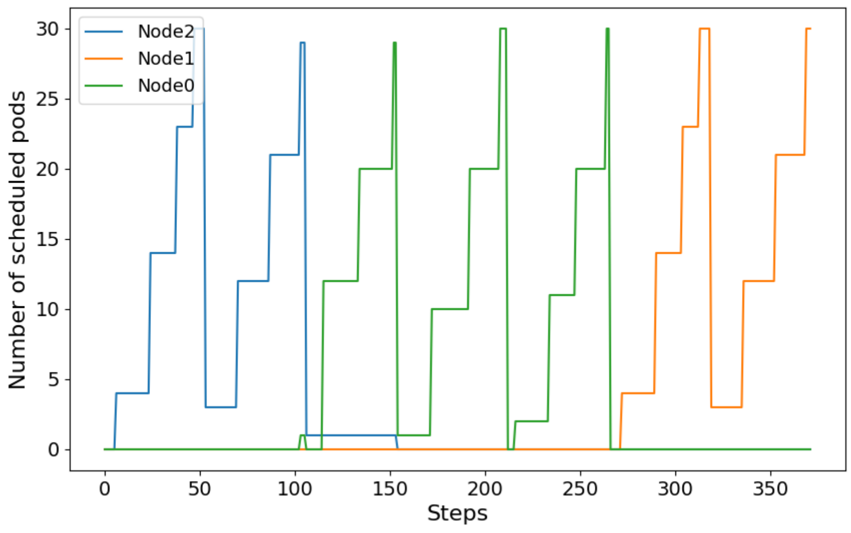
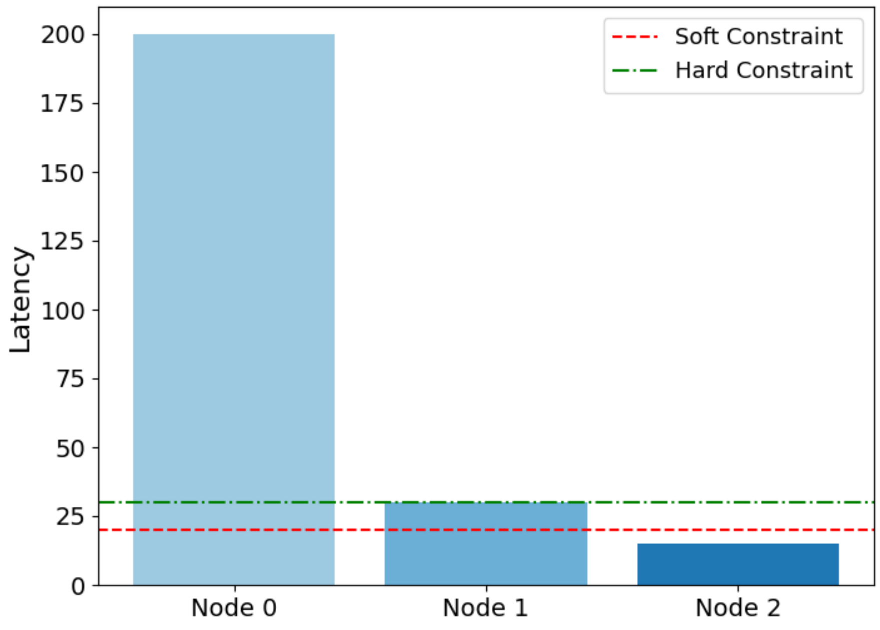


Figure 5.6: EL-RL Pods scheduling on node1 (test1)



Node2 with lower latency

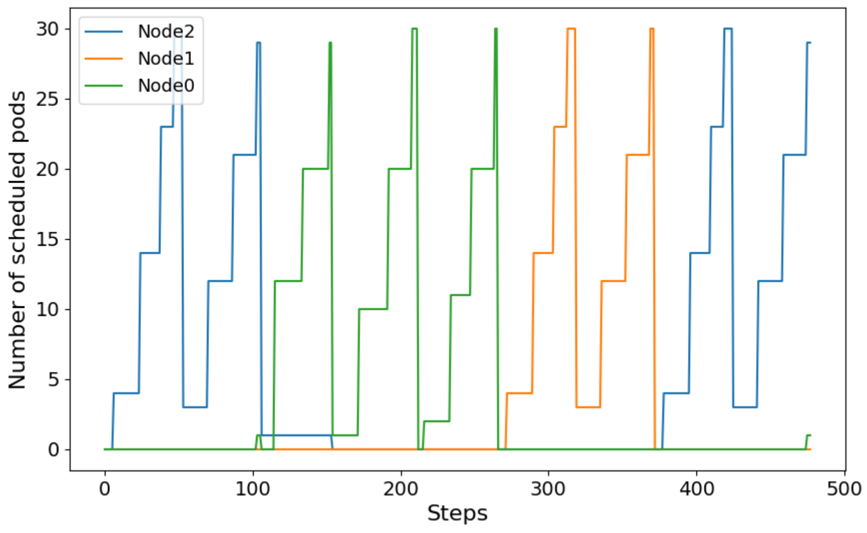


Figure 5.7: EL-RL Pods scheduling on node2 (test1)

The result obtained during testing is that the model can select the node with the lowest latency each time the latencies are varied. Additionally, the policy optimizes energy consumption by consistently suggesting the same node for scheduling new pods.

test2

The second test case further explores the model's response to changes in node latencies. This involves intentionally altering the latencies at two specific points in time to observe the model's adaptive behavior, as illustrated in the accompanying figure.

The test is modified to initially create 20 pods instead of 30.

The result obtained is:

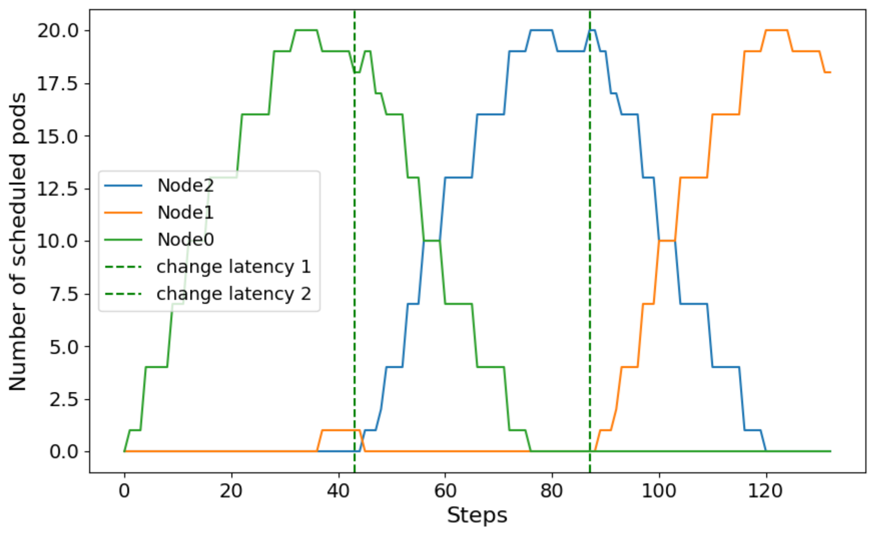


Figure 5.8: EL-RL Pods scheduling on nodes (test 2)

It is observed that the policy directs all the load to a single node. However, when latencies are modified, the model starts to shift the entire load to the best node to select based on the new latencies.

Such a dynamic adaptation is crucial in environments with fluctuating network conditions, ensuring optimal resource utilization and performance. The ability of the model to respond to latency changes can prevent potential bottlenecks and enhance the overall efficiency of the system. This adaptive approach can be particularly beneficial in cloud environments and distributed systems where network performance can vary significantly.

5.4 Comparisons between RL policies and default Kubernetes scheduler

Once the models were successfully trained and the agents effectively learned the intended policies by means of the reward mechanism, it became possible to compare RL policies with the default behavior of the Kubernetes scheduler by disabling the RL plugin when necessary.

Specifically, the comparison aims to highlight optimizations in terms of **energy consumption and latencies**.

Each model is tested individually, gathering data on:

- Number of active nodes at each step: the goal is to minimize the number of active machines. A machine is considered active when it as at least one pod scheduled.

- Chosen latency values at each step: the goal is to minimize latency over time.

Subsequently, the collected values are compared using a moving average approach to visually simplify the comparison of results.

Training phase comparison

At system startup, the agent's training phase begins. Initially, suggestions are random for all policies. This behaviour is similar to the one of the default scheduler that behaves as a load balancer. The following graph 5.10 shows how, *even before completing the actual training, the EL-RL agent makes better choices compared to other policies in terms of latencies.*

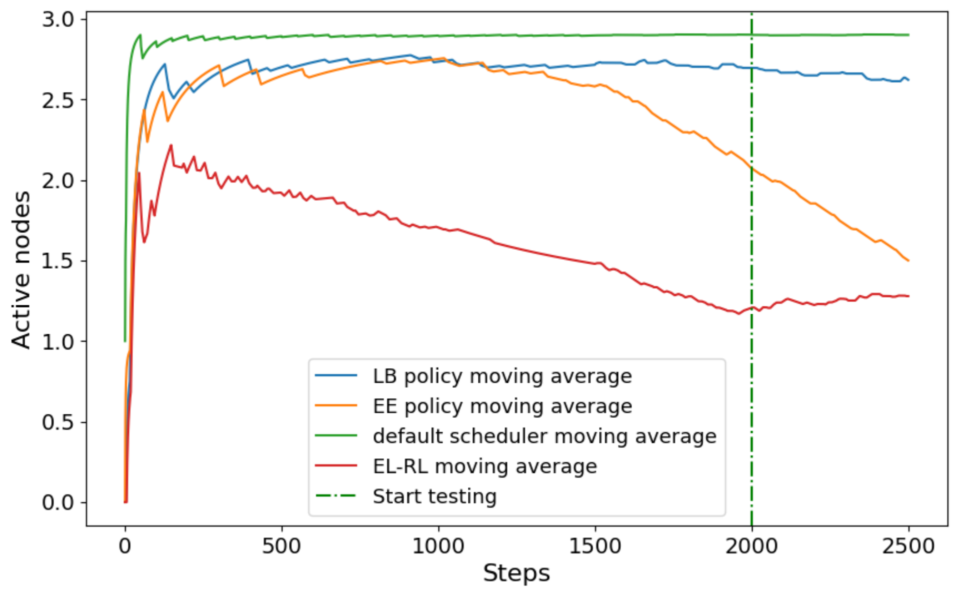


Figure 5.9: Comparison over time of active nodes values for different policies (training)

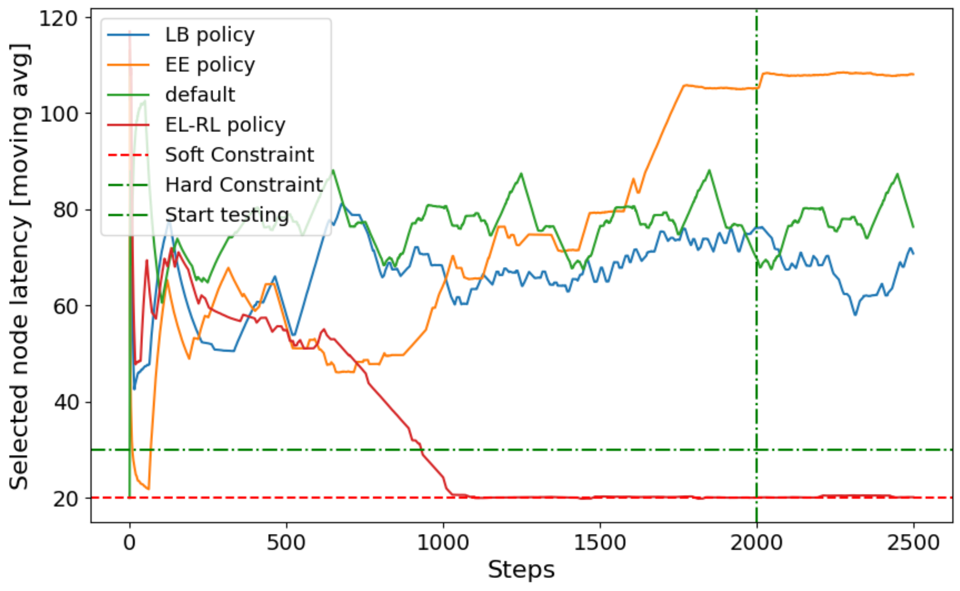


Figure 5.10: Comparison over time of latency values for different policies (training)

Testing phase comparison (Energy consumption)

The initial comparison obtained during the testing phase of the various models shows that the default Kubernetes scheduler tends to distribute the load across all available nodes by scheduling pods among all possible nodes, as seen in the case of the LB policy.

It is also observed that the EL-RL and EE policies exhibit a similar behavior, aiming to concentrate all the load on a single node to optimize energy consumption by minimizing the number of active nodes.

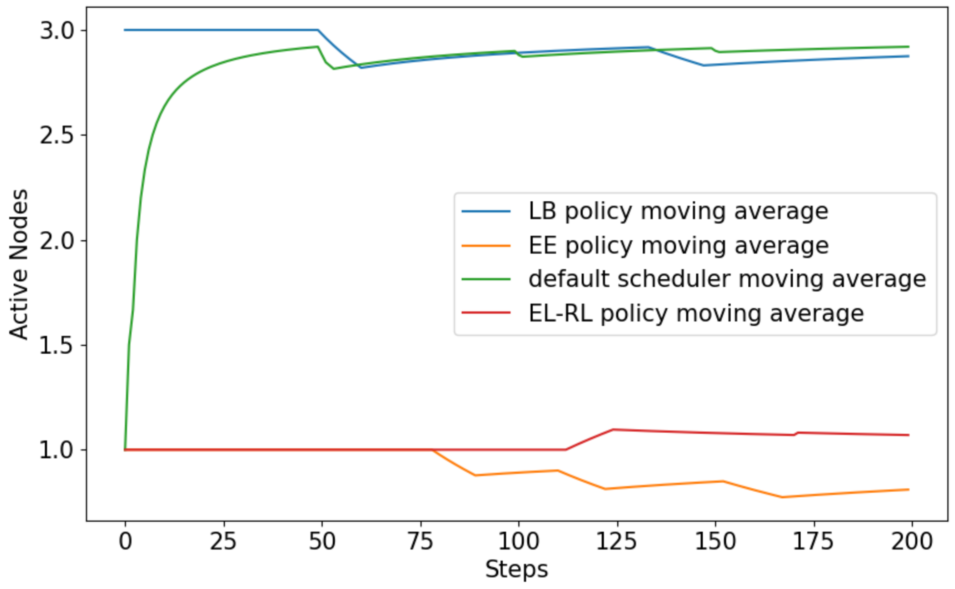


Figure 5.11: Comparison over time of active machines (testing)

Testing phase comparison (Node-user latency)

However, this second result highlights that, although the EE policy may be efficient in minimizing consumption, it does not take into account another important aspect: the latencies between nodes and the user, which are often critical for certain applications.

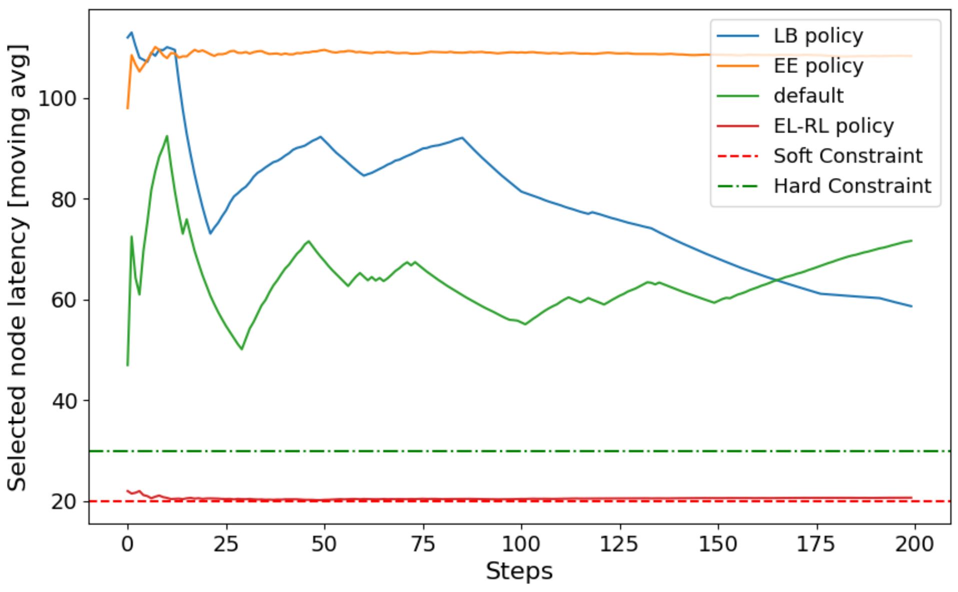


Figure 5.12: Comparison over time of latency values (testing)

Instead, the EL-RL policy not only optimizes energy consumption but also selects the node with lower latency that meets the latency requirements specified by the user.

In both cases, the proposed policy EL-RL performs better compared to the default behavior of the scheduler and the LB policy.

Moreover, in the EL-RL policy, the agent is trained to adapt to changes in the network. If an additional node offers lower latency, the workload is shifted to that node. Through the implementation of DeepSets and the use of masks, the agent accepts inputs regardless of permutations of values or reductions in input dimensions, allowing it to adapt to network changes.

Using RL enables flexible testing of various policies to improve scheduling based on different learned policies defined by a reward function.

Chapter 6

Conclusion

6.1 Overview

The primary objective of this thesis was to explore the possibilities offered by the combination of Kubernetes and Reinforcement Learning (RL) and to evaluate whether this union is advantageous compared to the use of the default scheduler provided by Kubernetes.

This research required the implementation of a simulated system, which, however, can be extended to real-world situations. The results obtained have demonstrated several advantages, including:

- Increased flexibility in the implementation of new scheduling policies, as the problem shifts to defining a new reward function.
- The possibility of achieving better results through the use of neural networks that can adapt and learn from environmental changes to determine the optimal behavior.
- The definition of standard policies obtained through RL.
- The definition of a new EL-RL policy guided by a model that optimizes scheduling in terms of energy consumption and latency.

In particular, the EL-RL policy implemented showed improvements over both the default scheduler and other policies initially hypothesized.

This thesis highlights the benefits of using AI algorithms and, thanks to the flexibility and composability of the systems used, serves as a foundation for potential extensions across various domains.

6.2 Future works

A potential future implementation is for example to extend the obtained results to a **multi-cluster environment**, leveraging the capabilities offered by tools such as **Liqo** [19].

Additionally, future research could explore the following directions:

- **Scalability and Performance Optimization:** Investigate the scalability of the RL-based scheduler in large-scale Kubernetes deployments. This includes testing the system under high load conditions and optimizing performance to ensure low latency and low energy consumption.
- **Integration with Other Scheduling Algorithms:** Explore the integration of RL-based scheduling with other advanced scheduling algorithms, such as those based on heuristics or machine learning, to create a hybrid scheduler that combines the strengths of different approaches.
- **Dynamic Reward Function Adjustment:** Develop methods for dynamically adjusting the reward function based on real-time feedback from the environment. This could improve the adaptability and responsiveness of the RL-based scheduler to changing workload patterns and resource availability.
- **Test and compare different RL algorithms** Implement and compare various RL algorithms to optimize performance.
- **Energy-Efficiency Enhancements:** Further refine the energy-efficient scheduling policy by incorporating additional factors such as thermal constraints, hardware-specific power consumption characteristics, and renewable energy sources. This could lead to even greater energy savings and environmental benefits.
- **Real-World Testing and Validation:** Conduct extensive testing and validation of the RL-based scheduler in real-world Kubernetes environments. This includes collaborating with industry partners to deploy the scheduler in production systems and gather empirical data on its performance and reliability.
- **User-Friendly Interfaces and Tools:** Develop user-friendly interfaces and tools to facilitate the configuration and monitoring of the RL-based

scheduler. This includes creating dashboards, visualization tools, and APIs that allow users to easily define reward functions, track scheduling decisions, and analyze system performance.

By pursuing these directions, the potential of combining Kubernetes with Reinforcement Learning can be fully realized, leading to more efficient, adaptable, and intelligent resource management in containerized environments.

6.3 Code repository

The code related to this implementation is publicly available on GitHub to encourage future collaborations and extensions. Specifically, the Git repository is located at [20].

Appendix A

System installation guide

In this appendix, we demonstrate how to recreate the system implemented in this thesis locally and how to make modifications to implement future work.

A.0.1 Prerequisites

The system requires the installation of components to run a kind cluster, including:

Software Requirements

- **Docker**: Install Docker to enable containerization and management of containers. Instructions for installing Docker can be found on the official Docker documentation.
- **kind**: Install kind (Kubernetes IN Docker) to create and manage local Kubernetes clusters using Docker containers. Installation instructions are available on the kind GitHub repository.
- **kubectl**: Install kubectl, the Kubernetes command-line tool, to interact with the Kubernetes cluster. Follow the kubectl installation guide for instructions.
- **Python**: Ensure Python is installed on your system. You can download it from the official Python website.
- **Go**: Install Go (Golang) as it is required for building and running certain Kubernetes components. Instructions for installing Go can be found on the official Go documentation.

Hardware Requirements

Ensure your system meets the following hardware requirements:

- **CPU**: At least 4 CPU cores.
- **Memory**: At least 8 GB of RAM.
- **Disk Space**: At least 20 GB of free disk space.

A.0.2 Installation

Download the repository from
`https://github.com/SoniaMatranga/SoniaMatrangaTesi`, which is structured as follows:

- **scheduler-plugins**: Contains the code for the new scheduler with the plugin that needs to be built into a new image and used instead of the default Kubernetes scheduler.
- **latency**: Contains the app to be deployed on the nodes to gather latency data.
- **model**: Contains the code for the RL agent.
- **venv**: Virtual environment with all necessary dependencies to run the model and integrate the custom environment.
- Configuration files.

Setup Kind Cluster

Create a Kind cluster with 4 nodes using the configuration file *kind-config.yaml*, ensuring the paths for the model and `venv` are correctly configured. Execute with:

```
kind create cluster --name vbeta3 --config kind-config.yaml
```

Note that the control node will mount model and `venv` volumes, where `venv` contains all the requirements requested by the cleanrl repository.

If it is needed it is possible to recreate the venv by installing cleanrl requirements, but in this case the gymnasium package must be modified by inserting and registering the custom environment for our model.

Configure Prometheus with Node Exporter

Install Prometheus configured with Node Exporter into the cluster to provide metrics to the RL model.

```
kubectl create namespace monitoring
helm repo add prometheus-community https://prometheus-
    community.github.io/helm-charts
helm repo update
helm install prometheus prometheus-community/prometheus-node
    -exporter -n monitoring
```

Follow the steps outlined in the article *kind-fix missing prometheus operator targets* to let Prometheus work on a Kind cluster.

Configure Custom Scheduler on Master Node

After automatically creating the cluster, add the configuration of the new scheduler into the master node (control-plane of the cluster). Add files to the following paths:

- `/etc/kubernetes/manifests`: Copy the `kube-scheduler.yaml` file containing the new configuration of the scheduler pod that will be created locally.
- `/etc/kubernetes`: Copy the `networktraffic-config.yaml` file containing the configuration of the scheduler plugins, from which you can enable or disable default behaviors.

Build and Load Local Scheduler Image

Generate the new scheduler image locally using the files from the *scheduler-plugins* directory, which contains the code for the plugins including the custom `NetworkTraffic` plugin. This plugin performs scoring based on the suggestions of the reinforcement learning model. Run the following command to generate a new Docker image:

```
make local-image
kind load docker-image --name vbeta3 localhost:5000/
    scheduler-plugins/kube-scheduler:latest
```

Alternatively, upload the image to DockerHub and modify the *kube-scheduler.yaml* configuration file to specify the image source for the master node.

Restart the Control Plane

Restart the control plane and delete the scheduler pod to create the new scheduler with the correct configuration and the local image. Retrieve the scheduler pod name using the command:

```
kubectl get pods -n kube-system
```

where the pod scheduler is named similarly to *kube-scheduler-vbeta3-control-plane*.

A.0.3 Scheduling Test

The `nginx-deployment.yaml` can be applied to the cluster to test scheduling, so that one pod is scheduled.

`kubectl create -f nginx-deployment.yaml`

Then, by observing the scheduler logs, it is possible to analyze the behavior when a new pod is scheduled:

```
   kubectl logs -f kube-scheduler-vbeta3-control-plane -n
 kube-system
```

The `deployments.sh` script will create multiple deployments, allowing you to train and test the model by executing it. This file will create the results shown in test1. Results shown in test2 cases can be obtained by executing `deployments2.sh`. To change the policy, it is possible to select one of the possible policies by modifying the custom environment and setting self-policy as indicated in the file *scheduling.py*.

A.0.4 Change policy

It is possible to modify the learned policy by setting the self.state variable in the scheduling.py file. It is crucial that when policy P0 (EL-RL) is set, the observation space is adjusted accordingly. Specifically, the code defining the observation space dimensions:

```
self.min_state = np.zeros((node_count()-1), dtype=np.float32
    )
self.max_state = np.full((node_count()-1), float('inf'),
    dtype=np.float32)
```

In policy P0, this needs to be changed to:

```
self.min_state = np.zeros((node_count()-1)*4, dtype=np.
    float32)
self.max_state = np.full((node_count()-1)*4, float('inf'),
    dtype=np.float32)
```

Failing to make this change will result in an error produced by CleanRL, as the dimensionality of the observation space will be incorrect when the `reset()` method is invoked.

A.1 Modify the system

A.1.1 Register a new custom environment

The first thing is to insert the custom environment file to the path

`$venv/lib/python3.9/site-packages/gymnasium/envs/classic_control`

Then it is needed to register the environment in gymnasium environment to let cleanrl use it. So in the file `__init__.py`, it must be added the line

```
from gymnasium.envs.classic_control.acrobot import
  YourEnvName
```

and in the file `envs/__init__.py` it must be added the env registration:

```
    register(
    id="YourEnvId",
    entry_point="gymnasium.envs.classic_control.yourenvfile:
YourEnvName",
    max_episode_steps=200,
    reward_threshold=195.0,
)
```

A.1.2 Modify the Reward Function

In order to test new policies, it is possible to define new reward functions by inserting the code into the **step function** of the custom environment.

When modifying `scheduling.py`, the environment defines `self.policy`, so from here a different policy can be set. In this case it is also needed to modify suggestion APIs and to insert the code to evaluate the state for the model.

A.1.3 Improving Neural Network Models

With models it is possible to modify:

- the hyperparameters used at the start of training, which are set in the plugin, specifically in the path scheduler–plugins/cmd/scheduler/main.go
- the neural network structure itself by modifying the path model/cleanrl/cleanrl/dqn.py. Clearly, it is also possible to test other agents available in the cleanrl library.
- the action selection during the training phase in model/cleanrl/cleanrl/dqn.py and during the testing phase by modifying the path: model/cleanrl/cleanrl_utils/evals/eval_dqn.py

For example, in this thesis, the used **hyperparameters** are:

Hyperparameter	**Value**
exploration-fraction	0.3
save-model	True
–total-timesteps	2000
–learning-rate	2.5e-3
–learning-starts	300

Table A.1: Table of model hyperparameters

For example, in this thesis, the test phase is modified from the one in cleanrl so that the function `predict` defined in our custom implementation is used.

```
        model = Model(envs).to(device)
    model.load_state_dict(torch.load(model_path,
map_location=device))
    model.eval()

    obs, _ = envs.reset()
    episodic_returns = []
    while len(episodic_returns) < eval_episodes:
        action_masks = []
        for env_idx in range(envs.num_envs):
            env = envs.envs[env_idx]
            if hasattr(env, 'action_masks') and callable(
getattr(env, 'action_masks')):
                action_mask = env.action_masks()
                action_masks.append(action_mask)
            else:
                action_masks.append(None)
            action_mask = action_masks[0]

        action_mask_matrix = action_mask.reshape(1, -1)
        if random.random() < epsilon:
            actions = []
            for env_mask in action_mask_matrix:
                valid_actions = np.where(env_mask)[0]
                actions = np.array([np.random.choice(
valid_actions) for _ in range(envs.num_envs)])
            #actions = np.array([envs.single_action_space.
sample() for _ in range(envs.num_envs)])
        else:
            #q_values = model(torch.Tensor(obs).to(device))
            #actions = torch.argmax(q_values, dim=1).cpu().
numpy()
            actions = model.predict(obs, action_mask_matrix)

        next_obs, _, _, _, infos = envs.step(actions)
        if "final_info" in infos:
            for info in infos["final_info"]:
                if "episode" not in info:
                    continue
                print(f"eval_episode={len(episodic_returns)
    }, episodic_return={info['episode']['r']}")
                episodic_returns += [info["episode"]["r"]]
        obs = next_obs

    return episodic_returns
```

A.2 Directory Structure and Key Files

Model Directory

- **model/cleanrl**: Contains all files of cleanrl that must be mounted on the scheduler pod.
- **model/cleanrl/cleanrl/dqn.py**: Modified DQN agent that uses DeepSets.
- **model/main.py**: Contains functions to expose agent suggestions and to allow communication between the virtual environment (venv) and Prometheus.
- Other files used in previous versions.

Scheduler-Plugins Directory

- **scheduler-plugins/build/scheduler/Dockerfile**: Dockerfile for the scheduler where new libraries are installed and the venv is activated.
- **scheduler-plugins/cmd/scheduler/main.go**: Starts the model and registers the custom plugin for the scheduler.
- **scheduler-plugins/pkg/networktraffic/networktraffic.go**: Defines the custom plugin.
- **scheduler-plugins/pkg/networktraffic/prometheus.go**: Functions for interaction between the custom plugin and Prometheus in case communication with the agent fails.

- Additionally, there are changes to library versions to ensure the repositories work correctly. See `scheduler-plugins/README.md` for more details.

Venv/lib/python3.9 Directory

- **/site-packages/gymnasium/envs/classic_control/scheduling.py** contains the definition of the environment used by the DQN agent, which is located in the model. It also starts an app to visualize graphs on the agent.

- **site-packages/gymnasium/envs/classic_control/___init___.py** registers the new custom environment in Gymnasium [18].

Configuration Files

- **kind-config.yaml**: Initial cluster configuration where Prometheus needs to be added.

- **kube-scheduler.yaml**: Configuration of the scheduler pod where volumes of plugins, model, and venv are inserted.

- **networktraffic-config.yaml**: Configuration file for the custom scheduler where plugins and default scheduler behaviors can be enabled or disabled. It passes Prometheus address as an argument.

A.3 Useful Commands

A.3.1 System

When starting the system, ensure that Docker is running. Then, by opening a new terminal, the scheduler logs can be viewed using the following command:

```
kubectl logs -f kube-scheduler-vbeta3-control-plane -n kube-
    system
```

In this command, the `-f` flag will follow the logs, providing real-time updates.

The system automatically generates graphs, which are exposed on port 8050 of the virtual cluster. To view these graphs on your local browser, you need to forward port 8050 of the virtual cluster to port 8050 on your localhost. This can be accomplished with the following command:

```
kubectl port-forward pod/kube-scheduler-vbeta3-control-plane
    8050:8050 -n kube-system
```

Initially, the graphs will be empty. To start the scheduling and populate the graphs, simply execute the deployment script:

```
./deployment.sh
```

A.3.2 Model

CleanRL offers the ability to monitor the learning progress of the model using **TensorBoard** [21]. To start TensorBoard, follow these steps:

First, enter the control plane by executing:

```
kubectl exec -it kube-scheduler-vbeta3-control-plane -n kube
   -system -- /bin/bash
```

Within the control plane, start TensorBoard by running:

```
tensorboard --logdir runs
```

Next, forward the TensorBoard service to be accessible from localhost:

```
kubectl port-forward pod/kube-scheduler-vbeta3-control-plane
    6006:6006 -n kube-system
```

After performing these steps, you can access TensorBoard on your local browser at `http://localhost:6006`, allowing you to visualize and monitor the learning process of your model.

Bibliography

[1] *Kubernetes Documentation: Concepts - Architecture*. Accessed: June 30, 2024. The Kubernetes Authors. 2023. URL: https://kubernetes.io/docs/concepts/architecture/ (cit. on pp. 1, 10, 11).

[2] J. Rothman and J. Chamanara. «An RL-Based Model for Optimized Kubernetes Scheduling». In: *2023 IEEE 31st International Conference on Network Protocols (ICNP)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2023, pp. 1–6. DOI: 10.1109/ICNP59255.2023.10355623. URL: https://doi.ieeecomputersociety.org/10.1109/ICNP59255.2023.10355623 (cit. on pp. 2, 7).

[3] Bjorn Rabenstein and Julius Volz. «Prometheus: A Next-Generation Monitoring System (Talk)». In: *Prometheus*. 2015 (cit. on pp. 2, 41).

[4] Joshua A. Achiam and contributors. *CleanRL: High-quality single-file implementations of Deep Reinforcement Learning algorithms*. https://github.com/vwxyzjn/cleanrl. Accessed: 2024-06-26. 2023 (cit. on pp. 3, 32).

[5] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander J. Smola. *Deep Sets*. NIPS 2017. 2017. DOI: 10.48550/arXiv.1703.06114. arXiv: 1703.06114 [cs.LG]. URL: https://arxiv.org/abs/1703.06114 (cit. on pp. 3, 24).

[6] Víctor Medel, Carlos Tolón, Unai Arronategui, Rafael Tolosana-Calasanz, José Bañares, and Omer Rana. «Client-Side Scheduling Based on Application Characterization on Kubernetes». In: Oct. 2017, pp. 162–176. ISBN: 978-3-319-68065-1. DOI: 10.1007/978-3-319-68066-8_13 (cit. on p. 6).

[7] Cong Xu, Karthick Rajamani, and Wesley Felter. «NBWGuard: Realizing Network QoS for Kubernetes». In: *Proceedings of the 19th International Middleware Conference Industry*. Middleware '18. Rennes, France: Association for Computing Machinery, 2018, pp. 32–38. ISBN: 9781450360166. DOI: 10.1145/3284028.3284033. URL: https://doi.org/10.1145/3284028.3284033 (cit. on p. 6).

[8] Ali J. Fahs and Guillaume Pierre. «Tail-Latency-Aware Fog Application Replica Placement». In: *Service-Oriented Computing: 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings*. Dubai, United Arab Emirates: Springer-Verlag, 2020, pp. 508–524. ISBN: 978-3-030-65309-5. DOI: 10.1007/978-3-030-65310-1_37. URL: https://doi.org/10.1007/978-3-030-65310-1_37 (cit. on p. 6).

[9] Nicola Di Cicco, Gaetano Francesco Pittalà, Gianluca Davoli, Davide Borsatti, Walter Cerroni, Carla Raffaelli, and Massimo Tornatore. «DRL-FORCH: A Scalable Deep Reinforcement Learning-based Fog Computing Orchestrator». In: *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. 2023, pp. 125–133. DOI: 10.1109/NetSoft57336.2023.10175398 (cit. on pp. 7, 8, 32).

[10] Amazon. *Renewable Energy*. Website. Retrieved 2021-02-01. URL: https://sustainability.aboutamazon.com/environment/sustainable-operations/renewable-energy?energyType=true (visited on 02/01/2021) (cit. on p. 8).

[11] Hölzle, Urs. *Data centers are more energy efficient than ever*. Website. Retrieved 2021-02-01. URL: https://blog.google/outreach-initiatives/sustainability/data-centers-energy-efficient (visited on 02/01/2021) (cit. on p. 8).

[12] Aled James and Daniel Schien. «A Low Carbon Kubernetes Scheduler». In: July 2019 (cit. on p. 8).

[13] Kuljeet Kaur, Sahil Garg, Georges Kaddoum, Syed Hassan Ahmed, and Mohammed Atiquzzaman. «KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem». In: *IEEE Internet of Things Journal* 7 (2020), pp. 4228–4237. URL: https://api.semanticscholar.org/CorpusID:202765604 (cit. on p. 8).

[14] Isabelly Rocha, Christian Göttel, Pascal Felber, Marcelo Pasin, Romain Rouvoy, and Valerio Schiavoni. «Heats: Heterogeneity-and Energy-Aware Task-Based Scheduling». In: *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP).* 2019, pp. 400–405. DOI: `10.1109/EMPDP.2019.8671554` (cit. on pp. 8, 9).

[15] Julio Renner. *K8s - Creating a Kube Scheduler Plugin.* Accessed: 2024-06-27. 2023. URL: `https://medium.com/@juliorenner123/k8s-creating-a-kube-scheduler-plugin-8a826c486a1` (cit. on pp. 12, 31).

[16] Hugging Face. *An Introduction to Deep Reinforcement Learning.* `https://huggingface.co/blog/deep-rl-intro`. Accessed: 2024-07-02. 2023 (cit. on pp. 17, 18).

[17] Kubernetes SIGs. *scheduler-plugins.* Accessed: 2024-06-27. 2024. URL: `https://github.com/kubernetes-sigs/scheduler-plugins` (cit. on p. 31).

[18] *Gymnasium by Farama.* `https://gymnasium.farama.org/index.html`. Accessed: 2024-07-09 (cit. on pp. 35, 71).

[19] *Liqo.* Accessed: 2024-07-09. URL: `https://liqo.io/` (cit. on p. 61).

[20] Sonia Matranga. *Sonia Matranga's Thesis GitHub Repository.* `https://github.com/SoniaMatranga/SoniaMatrangaTesi`. Accessed: 2024-07-09 (cit. on p. 62).

[21] TensorFlow. *TensorBoard.* `https://www.tensorflow.org/tensorboard?hl=it`. Accessed: 2024-07-09. 2024 (cit. on p. 72).