National College of Ireland

Comparative Study of RL Algorithms for Resource Optimization Scheduling in Kubernetes

MSc Research Project
Cloud Computing

Jay Shukla
Student ID: X23113111

School of Computing
National College of Ireland

Supervisor: Prof. Punit Gupta

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Jay Milind Shukla
Student ID:	X23113111
Programme:	MSc. Cloud Computing
Year:	2023-24
Module:	MSc Research Project
Supervisor:	Prof. Punit Gupta
Submission Due Date:	12/08/2024
Project Title:	Comparative Study of RL Algorithms for Resource Optimization Scheduling in Kubernetes
Word Count:	2,128
Page Count:	22

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Jay Shukla
Date:	12-08-2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Comparative Study of RL Algorithms for Resource Optimization Scheduling in Kubernetes

Jay Shukla
23113111

Abstract

Cloud computing has evolved from using monolithic architectures to relying on microservices. As microservices have become more common, managing them has also become more straightforward. Tools like Kubernetes, Docker, and OpenShift make it easier to deploy applications in containers, which helps reduce costs and save resources compared to older, monolithic systems. However, one challenge with microservices is that auto-scaling methods often treat each service individually, without considering how they interact with each other. This can lead to inefficient scaling, where either too many resources are used or not enough, potentially harming application performance. This paper suggests a new approach by using Reinforcement Learning (RL) algorithms alongside Kubernetes' Horizontal Pod Autoscaler (HPA) to improve how resources are managed and scaled. By doing so, we can better optimize performance and resource use in complex, dynamic microservice environments, ultimately improving application efficiency and reducing costs.

1 Introduction

Reinforcement learning (RL) has recently emerged as a promising technique for autonomous systems that make intelligent decisions about the environment. The thesis of this paper is an account of research on RL focused primarily on those who describe the process and determine its effectiveness. It especially focuses on training RL models based on indicators such as FPS, frequency, and elapsed time. Training success is measured by the model's Kulbach-Leibler deviation, split distribution, entropy loss, explained variance, learning rate, and loss set, which fully describe the model's performance. That calmness.

The learning process in RL has issues of computational efficiency and plays an important role in the implementation of RL, especially in areas where resource constraints and costs are major concerns. To answer this question, this paper uses the GYM Horizontal Pod Autoscaler (HPA) algorithm on a Kubernetes cluster. Using Kubernetes, an open source software for deploying and managing applications, RL is ideal for rapid development from a training environment. The GYM HPA algorithm runs predictively on a Kubernetes cluster and provides self-sufficient predictions to determine which pods are running based on real-time characteristics such as CPU usage, memory usage, and output usage.

The GYM HPA algorithm monitors each node's resource usage and increases or decreases the number of active shells at any time. Power adjustment is important for AC

which is common in RL training, where the resource consumption can quickly slow down the machine depending on the complexity of the task and the training phase so by using HPA the system ensures resource allocation better because of improving the performance of the J. and eliminating costs.

During the build process, separate reports are generated that track the progress and resource usage of the RL model on your Kubernetes cluster. Continuously monitor KPIs, including frames per second (fps), clicks, training performance, time spent and total steps. We also consider performance metrics such as Kullback-Leibler (KL), threshold level, entropy loss, observation variance, learning rate, total loss, number of abstractions, lack of design constraints, lossy design definition, and model variables.

Keeping a close eye on resource, CPU and memory usage metrics. These parameters are important for determining information about the HPA detection algorithm. For example, excessive CPU or memory usage can increase the number of HPA threads, causing load sharing and performance bottlenecks. On the other hand, using lesser number of HPAs reduces the number of changes, saves resources and reduces costs.

Integrating the GYM HPA algorithm into a Kubernetes cluster not only improves the performance and cost-effectiveness of RL training but also provides a reliable framework for heavy workloads. This process helps the resources to be regularly reviewed and adjusted accordingly to meet the changing environment requirements.

In Summary this paper presents a detailed evauation of RL being used to optimise the kubernetes enviroment by means of the GYM HPA algorithm. By means of this research our goal is to increase the understanding of the RL tool and contribute towards its development. THis work shows the ability to combine the GYM HPA algotrithm to create more simple, scalable and efficient algorithms.

1.1 The Significance of Container Scheduling

Cloud computing is a form of delivery that enables some of the most prominent technological services such as IaaS, PaaS, and SaaS without having actual hardware support. This shift has been made possible by Virtualization technologies; technologies that allow production of configurations that are virtual copies of actual physical hardware. This means we can now create platforms that act as a physical system in the digital space. Arguably one of the largest advancements in recent times have been the birth of the container, a light-weight package that comprises of everything a software application needs to function.

This has however, been made a norm in the recent past years. There are tools that handle the containers such as docker and kubernetes through which the arrangement, mobilization and control of containers is done. This makes it easy to schedule, deploy and scale since the life cycle time for each is much easier. Of these tools, Kubernetes is the most favored due to the existence of numerous features and conveniently applied, which can be regarded as the reason for the rapidly growing popularity.

Cloud computing has shifted from large monolithic applications to microservices applications to ease use. These have become easier top work with by virtue of Kubernetes or Docker or Open Shift and what it does is that each part of an application runs in a container. Compared to the traditional methods and the monolithic application, this approach is cheaper and requires fewer resources. However, one of the problems that come with its implementation is that most auto-scaling and scheduling methods work independently for each micro-service. They usually ignore the way these small services

are interconnected and this results in inefficiencies.

Every contemporary cloud and edge computing system requires container scheduling. The other thing is flexibility of the container scheduling to use the resources optimally for the dynamic computer systems that are in use these days. Resource management specific to contains assists with the saving of resources such as CPU, memory, and storage by scheduling the running of containers according to time of the day. In the case of edge clouds, where resources may be limited and have to be optimally partitioned used to retain the application's speed and flow, dynamic distribution is very useful. Optimal scheduling is cost saving, since there will not be need to employ many people, thus reducing on the cost of paying employees, and enhances the efficiency of the infrastructure in as much as it deters over-provision and under-utilization. Another advantage is that far as genericity, container scheduling also improves system scalability and dependability. Redundancy and load balancing can leverage containers in schedule the containers in many nodes and clusters applying complex algorithms. This distribution enhances reliability since no single node of the given system can fail and make the entire system inaccessible. In addition, container scheduling allow for the scalability in horizontality, which means question those systems are capable of increase or decreasing the number of containers according to the traffic they have to handle. This function helps to maintain the stability of the system and users' satisfaction during the periods of demand by maintaining the availability of applications.

1.2 Research Objective

The aim of this research project is to create a reinforcement learning (RL) agent that works with the gym-hpa algorithm, inspired by the Gym framework, to optimize resource usage in Kubernetes environments. The focus on implementing a single RL agent to reduce resource consumption and explore various work scheduling strategies to improve the performance of pods and containers. By using RL algorithms, we hope to enhance the resource optimization capabilities of Kubernetes, making it more effective at managing systems with multiple components. So the the main question driving this research is: **"Can RL agent improve Kubernetes auto-scaling for multicomponent systems?"**

1.3 Report Organization

1. **Related Work**: This section reviews the existing literature on reinforcement learning algorithms and their application in Kubernetes scheduling, with a focus on auto-scaling in fog and edge computing environments.

2. **Methodology**: This section details the setup and methodology followed in implementing the custom environments using the Gym library, focusing on CPU, memory, and cost optimization strategies in Kubernetes.

3. **Design Specification**: This section discusses the design and specifications of the systems and algorithms used, including the integration of the HPA algorithm with Kubernetes.

4. **Implementation**: This section describes the implementation process, tools, and technologies used, along with the challenges faced and the solutions adopted.

5. **Evaluation**: This section presents and analyzes the results of the experiments conducted to evaluate the effectiveness of the implemented solutions in optimizing resource usage.

6. **Discussion**: This section provides a detailed discussion of the experimental findings, critiques the design, and suggests possible improvements and future research directions.

7. **Conclusion and Future Work**: This final section restates the research objectives, summarizes the key findings, discusses the implications, and proposes meaningful future work to extend the research.

2 Related Work

The report which have chosen, "Auto-scaling Policies to Adapt the Application Deployment in Kubernetes" Rossi (2020), elaborates the innovations in procedures aimed at increasing the possibility of auto-scaling in Kubernetes, a platform that is commonly used to manage applications based on containers. Originally, Kubernetes has implemented an HPA which was based on fixed threshold policies and which had mainly CPU usage as a metric for scaling applications. However, this approach often struggles to meet the Quality of Service (QoS) needs of applications, especially those sensitive to latency, because setting the right thresholds is complex and varies from one application to another.

To overcome these challenges, the report suggests using reinforcement learning (RL) to create more adaptive and efficient auto-scaling policies. This is unlike other methods of thresholding, for example, the fixed-scaling thresholding where the scaling is adjusted based on the observed data and the specific target QoS such as response times, low. From the work of the scholars, it can be deduced that the model-based RL policies provide higher efficacy compared to the model-free counterpart which suggests that adjustment in the organisational scaling takes place in a quicker and more efficient manner. This research also reveals that when adopted into the context of Kubernetes, such RL-based policies enhance the platform utilization to meddle with the customers' dynamic workloads, hence enhancing the service availability and efficiency in cloud environments.

The existing mechanisms of Kubernetes auto-scaling, for example, HPA, are threshold-based mainly on CPU usage figures. Though these methods are quite useful in managing specific loads, they do not deliver the Quality of Service (QoS) needed by low-latency applications. While establishing the thresholds that fit the particular case is a complex task that often calls for professionals' decisions and constant tweaking, tuning is not always effective and optimized, specifically when it comes to the dynamic environments of clouds.

To overcome these limitations, recent work has shifted towards using machine learning, specifically reinforcement learning (RL), for generating more intensive and effective auto-scaling policies Pramesti and Kistijantoro (2022). Therefore, RL agents should make their decisions on the basis of current demands for resources and the QoS requirements, since RL agents are built to learn from real-time data. From present studies, performance boosts incorporating application response times and resource utilization are widely observed, and based on these implementations, it becomes a more reliable and self-sufficient

solution for addressing new and varying workloads in microservice deployments from the cloud.

The paper proposes a novel auto-scaler for microservices deployed in Kubernetes, where the emphasis is placed on forecasting response time in order to achieve better scaling Pramesti and Kistijantoro (2022). Originally, Kubernetes employs the Horizontal Pod Autoscaler (HPA) which depends mostly on the CPU utilization to determine the scaling conditions. However, this new approach uses a machine learning model to determine how many hours it will take to get a response from a system by using performance measures from both the microservice and node levels. All these predictions can help the auto-scaler to determine the required number of pods in order to maintain the response time within the set range. The results suggest that this method achieves a lower response time compared to the traditional HPA, though it consumes more resources. This shows how machine learning can help enhance scaling in Kubernetes and thus enhance applications' responsiveness and effectiveness.

2.1 RL Algorithm

Reinforcement learning is an effective strategy concerning autonomic adaptation for cloud services due to the fact that it can do auto-scaling based on the dynamic behavior of the cloud Tesauro et al. (2006). Some of the initial RL applied were Q-Learning and SARSA, which were aimed at server and virtual machine scheduling. Tesauro et al. (2006) and Horovitz and Arian (2018) also used RL for virtual machine management. In order to solve this problem, more recently RL methods have been used for container scheduling. Horovitz and Arian (2018) developed an RL technique for updating the scaling thresholds by enhancing state space reduction. Rossi et al. (2019) was a continuation of enhancing RL methods through integrating system knowledge into the containers' scheduling decision-making process.

The A-SARSA algorithm, introduced by Zhang et al. (2020), analyzes the existing problems with using traditional reinforcement learning methods for container scheduling. Q-Learning and SARSA algorithms, which are traditional RL methods, end up having high SLA violation due to problems such as scheduling at the wrong time and decision-making issues. A-SARSA improves on the basic SARSA algorithm by adopting the ARIMA model for workload prediction and an ANN for CPU utilization and response time prediction. This way, the predictability and accuracy of scaling strategies are guaranteed, SLA violation rates considerably decreased, but at the same time, the application keeps a good level of resource utilization.

According to the study Gradient and Gradient (2020), the DDPG algorithm is presented as a very effective reinforcement learning model that combines two methods: Q-learning and Policy Gradient methods. It is perfect for complete actions as opposed to discrete ones, in environments that do not presuppose constant interruption. In the field of Kubernetes auto-scaling, DDPG can be employed for making the best scaling strategies by learning from the environment and changing the number of allocated resources in real-time.

The Deep Deterministic Policy Gradient (DDPG) algorithm is an advanced reinforce-

ment learning technique that combines the benefits of Q-learning and policy gradient methods, making it suitable for environments with continuous action spaces. This algorithm is particularly effective for tasks that require precise and continuous control, such as autonomous driving.

The main method that the DDPG Algorithm uses is the actor-critic method. The actions to be taken by the actor network include the provision of more or less resources, while the critic network measures the worth of these actions by estimating them. This setup makes learning tangible, stable, and also efficiently delivered throughout the learner's active academic experience. Sharing this, DDPG applies experience replay and target networks for the purpose of improving the stability of learning. Experience replay is a technique in which past actions are saved and occasionally selected from so as to disrupt the flow of events, making learning more consistent. The target networks that are updated more slowly furnish stable criteria for evaluation, and they contribute to stabilizing the learning process. These features make DDPG suitable especially for Kubernetes auto-scaling and aid in making ones that are adaptable to new conditions.

The article Esfandiari and Atashgah (2022) shows the efficiency of PPO for control problems, especially in environments with continuous action space. For the most part, their work pertained to robotics and autonomous systems; however, if one isolates the applicable components, a very natural extension of PPO techniques can be made to auto-scaling and scheduling problems in cloud-based infrastructures. The high-performance learning features of PPO, on the other hand, are mainly the stability of learning as it limits the degree of policy updates; unfavourable adjustments that can affect the program significantly. This feature is especially beneficial in technically volatile environments such as cloud computing where standardization and standard-based services delivery is a core success factor.

In regard to auto-scaling specifically, PPO can be used to optimize resource distribution because of its ability to learn from changes in workload. Since it entails motions on-scaling of assorted applications by processing state information from a policy network range, PPO can solve the complex problem of handling application scaling in Kubernetes. The episodic vs continuous state and action space also allow for fine-tuning of the resources that are to be provided to the applications to get the desired output while not being over-provisioned. Therefore, this capability puts the PPO algorithm in a place to be a promising approach in developing adaptive and responsive auto-scaling strategies in cloud environments that will actively improve efficiency and performance

When I was looking for ways to improve how our work uses resources, I came across this really interesting article Silva et al. (2020). The authors introduced a new approach for adjusting resources automatically in Kubernetes. Usually, when apps have different parts that work together, the way resources are shared doesn't take that into account. But their method uses what they call "hierarchical resource management" to make better choices about resources while thinking about how the whole app is structured.

2.2 Table of the RL Algorithms

Table 1: RL Algorithms and Their Applications in Cloud, Kubernetes, and Containers

Algorithm	Description	Application Area	Cloud Services
Q-Learning Ahmed and Ammar (2017)	Learns the value of each action in a state-action pair to find the best action.	Cloud optimization, Kubernetes	AWS SageMaker, OpenAI Gym
Deep Q-Network (DQN) H. Mao and Kandula (2016)	Uses deep neural networks to approximate the Q-value function.	Cloud resource management, Kubernetes	Google Cloud AI Platform, AWS SageMaker
SARSA F. Zhang and Xu (2017)	Updates the Q-value based on the action actually taken.	Cloud resource management	Theoretical applications
DDPG K. Zhang and Yuan (2019)	Combines Q-learning and policy gradient for continuous action spaces.	Kubernetes auto-scaling, Cloud control	AWS SageMaker RL, Azure ML
A3C X. Chen and Bennis (2018)	Runs multiple parallel instances to stabilize training.	Kubernetes, Distributed clouds	AWS, Microsoft Azure
TRPO X. Jin and Reisslein (2018)	Ensures safe policy updates using a trust region.	Cloud resource management	Google Cloud, AWS SageMaker
Dueling DQN R. Li and Zhang (2018)	Separates state value and advantage function for better learning.	Kubernetes, Containers	AWS SageMaker, Google AI
Actor-Critic T. P. Lillicrap and Tassa (2017)	Combines policy and value functions to optimize the policy.	Cloud optimization	Various cloud platforms
REINFORCE S. Gupta and Chaturvedi (2020)	Follows the gradient of expected rewards for updates.	Theoretical cloud applications	Academic settings

3 Methodology

Earlier, the RL algorithms employed in cloud environments were discussed with regards to its subcategories. Next, let's discuss the details of the HPA algorithm and its integration with the certain Platform and applying on dynamic applications and why choose the RL algorithm.

This RL-based algorithm is based on Gym where we can train an agent that is trained in how to efficiently utilize resource in the future time period. While other machine learn-

ing techniques could potentially be applied to HPA, reinforcement learning is particularly well-suited for this type of dynamic resource allocation problem. RL allows the system to learn optimal scaling policies through trial-and-error interactions with the environment, adapting to changing workloads over time. This aligns well with the unpredictable nature of pod scaling in Kubernetes clusters.

3.1 Gym-HPA Architecture

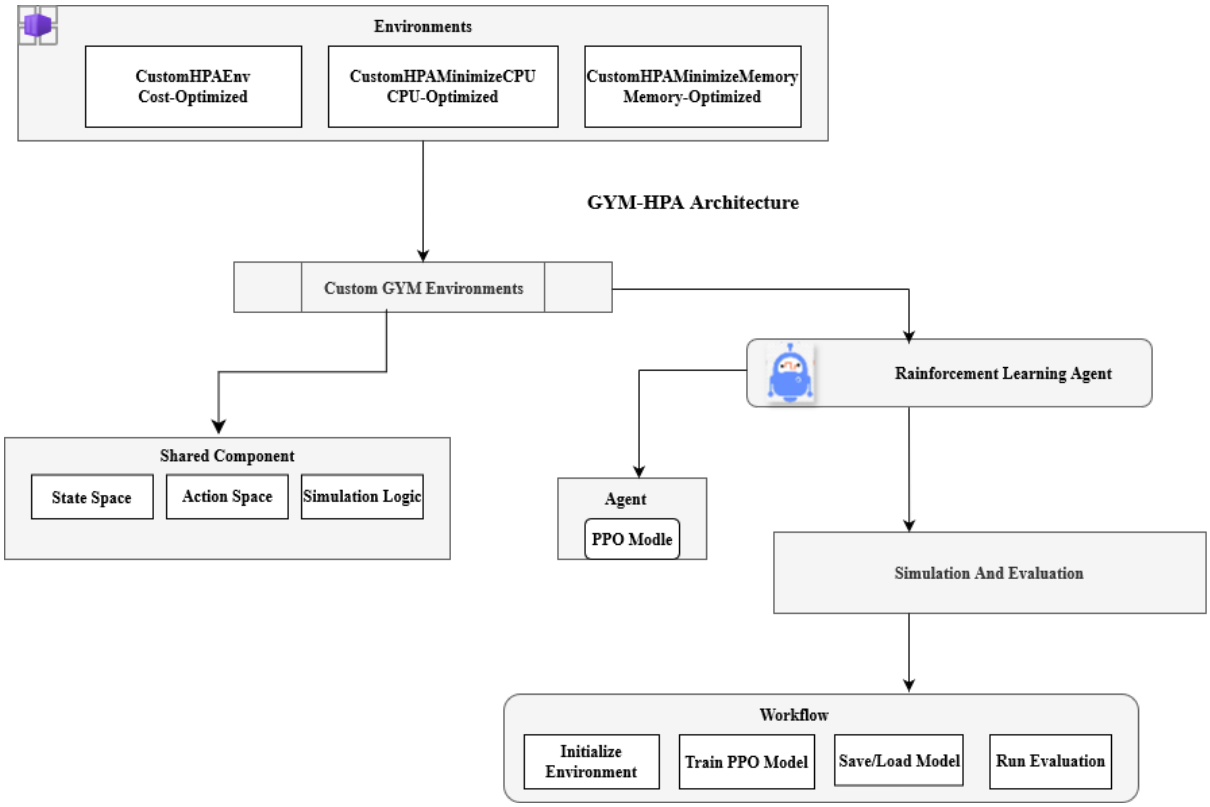

Figure 1: GYM-HPA Architecture

The GYM-HPA architecture is organized into three main sections:

1. **Custom Gym Environments:**
   - These environments are specifically designed to optimize different metrics like CPU usage, memory usage, and overall cost.
   - Each environment is a Python class that builds on `gym.Env` and includes functions for initializing the environment, resetting it to the start, executing actions, and displaying the current state.
2. **Reinforcement Learning Agent:**
   - The agent utilizes the Proximal Policy Optimization (PPO) algorithm, provided by the Stable Baselines3 library, to interact with the environments.
   - It learns the best strategies for scaling Kubernetes pods based on the resource metrics.

3. **Simulation and Evaluation Workflow:**
    - This section outlines the process, starting from setting up the environment, training the PPO model, saving or loading the model, and finally running evaluations to test the effectiveness of the learned strategies.
4. **Cost Optimization Environment:**
    - Designed to minimize total costs by balancing both CPU and memory usage.
5. **CPU Optimization Environment:**
    - Focuses on reducing the CPU usage across Kubernetes pods.
6. **Memory Optimization Environment:**
    - Aims at minimizing memory usage for each pod.
7. **State Space:**
    - This represents the current resource usage and the number of pods as a 2D numpy array.
8. **Action Space:**
    - The agent has three possible actions—scale down, keep the number of pods the same, or scale up—depending on the current state.
9. **Cost Optimization:**
    - Calculates the total cost based on the number of pods and their resource consumption.
10. **CPU Optimization:**
    - Specifically targets reducing the CPU usage per pod.
11. **Memory Optimization:**
    - Focuses on minimizing memory usage per pod.

This architecture is designed to be flexible, making it easy to experiment with and compare different Horizontal Pod Autoscaling (HPA) strategies within a simulated Kubernetes environment. This setup allows for a systematic evaluation of how various optimization approaches perform in managing resources like CPU and memory in a cloud environment.

3.2 Proximal Policy Optimization (PPO) Algorithm

Y. Sun and Wang (2019) in their paper "Model-Based Reinforcement Learning via Proximal Policy Optimization" define PPO as one of the most powerful and widely used reinforcement learning algorithms owing, in part, to the balance between simplicity and efficacy. As mentioned in the paper, PPO belongs to the policy gradients methods and improve the policies with multiple epochs of stochastic gradient ascent, in this way, the update avoids drift far away from the current policy. In this regard, PPO proves more useful in cases with continuous action spaces contributing to the general stability of learning.

Specifically, to further incorporate IL in the context of the given content, In a model-based RL framework that uses Gaussian Process Regression (GPR) for dynamic model learning and Proximal Policy Optimization (PPO) for policy improvement. As described by the authors in their paper, this integration ofGPR enables the model to control the uncertainty of environment by predicting the dynamic change and adjust the policy subsequently.

Algorithm 1 High level overview of PIPPO

0: **init:** Apply random control signals to the environment and record training data;
0: **repeat**
0: Learn probabilistic dynamic model *gp* by Gaussian process regression using all data, see Section A;
0: **repeat**
0: Sample state s_t using particle technology and propagate to s_{t+1} based on *gp*, see Section B;
0: Policy improvement based on PPO, see Section C;
0: **until** convergence
0: **return** π^*
0: Apply π^* to the environment and record data;
0: **until** task learned =0

- **init:** The algorithm starts by applying random control signals to the environment to gather initial training data.
- **repeat:** The algorithm enters a loop to continuously update the dynamic model and policy.
- **Learn probabilistic dynamic model *gp*:** Gaussian Process Regression (GPR) is used to build a probabilistic model of the environment's dynamics.
- **repeat:** A nested loop begins to refine state sampling and policy improvement.
- **Sample state s_t and propagate to s_{t+1}:** States are sampled using particle filtering, and the next state is predicted using the *gp* model.
- **Policy improvement based on PPO:** The policy is optimized using Proximal Policy Optimization (PPO) based on the predicted states.
- **until convergence:** This inner loop continues until the policy stabilizes and converges.

- **return π^*:** The optimized policy π^* is returned after convergence.
- **Apply π^* to the environment:** The optimized policy is applied to the environment, and the results are recorded.
- **until task learned:** The outer loop continues until the task is fully learned and the desired performance is achieved.

3.3 Custom Environment Setup for Resource Optimization

The implementation involved creating three custom Python environments using the Gym library, each utilizing the Proximal Policy Optimization (PPO) algorithm to optimize different aspects of resource usage in a Kubernetes cluster.

1. **Memory Optimization:** The first environment is designed to reduce the memory consumption for Kubernetes pods. The RL agent in this environment takes as input the current state information, which includes the number of pods and the amount of resources they use. It then decides whether to scale up (add more pods), scale down (reduce the number of pods), or maintain the current number of pods. The reward function in this setting is designed to discourage the agent from consuming excessive memory, thereby encouraging efficient memory usage.
2. **CPU Optimization:** The second environment seeks to reduce CPU demand. Similar to the memory optimization environment, the RL agent observes the CPU utilization of the pods and performs actions to regulate the number of pods. The reward function in this environment penalizes high CPU usage, forcing the agent to learn strategies that keep CPU utilization low while maintaining application responsiveness.
3. **Cost Optimization:** The third environment aims to minimize a cost function that considers both the time complexity of the algorithm and the memory consumption. The cost is defined as a function of the number of active pods and the resources used by each pod. The RL agent's primary goal is to maximize resource efficiency while minimizing the cost associated with over-provisioning or under-provisioning resources.

3.4 Environment Parameters

In the context of the cloud and in a Kubernetes cluster, the utilization of elements such as CPU and memory should be closely managed to ensure that the cluster and hence the application performs optimally whilst at the same time not using up a lot of resources. To do this, simulations are used to study and analyze the behavioral pattern of the HPA for different scenarios. There are three major parameters that regulate these simulations, and both resources and scale within the Kubernetes environment based on those parameters. These parameters are categorized into four main areas: In addition to environment parameters, the RL algorithm parameters, the values of simulation metrics, and the specifications of the Kubernetes environment create the global parameter vector. All the categories perform a significant role in achieving the efficiency of the system and attaining the intended performance goals.

1. Environment Parameters

- State Space: The state space usually consists of the current resource usage statistics which include the CPU and memory usage in the pods of the Kubernetes cluster. The state space conveys information about the environment's conditions to the agent to make the appropriate scaling determinations.

- Action Space: The state space delineates the possible states of environment and the action space describes the possible actions that the RL agent can perform. In the context of Kubernetes, it is implied that such actions are usually scaling up, down, or keeping steady the number of pods. The formation of the action space is significant as it shapes the agent's decision-making process in choosing efficient strategies for resource utilization.

2. RL Algorithm Parameters

- Reward Function: Reward function is another important factor that sets the course of learning for the RL agent. The overall structure of the reward function is to lower values for undesirable phenomena (for example, high resource consumption or cost) and to increase values for desirable phenomena (for example efficient consumption of resources). The specific details of the reward function depend on what is to be maximized, memory, CPU or cost.

- Learning Rate: The learning rate determines the extent to which the knowledge acquired by the RL agent is revised, given new information. This may increase the learning rate to obtain faster learning, though this leads to instability; a small learning rate yields a slower learning procedure that is, however, more stable.

- Discount Factor (Gamma): The discount factor measures the level of preference for the future rewards in comparison with immediate rewards by the agent. A value bigger than 1 makes the agent look forward to the future, and less than 1 makes it consider the present.

- Policy Update Frequency: This parameter defines when the RL agent wants to update the policy of the agent, which is the decision rule used by the RL agent in an attempt to make accurate decisions on the state of the environment. The agent can be updated frequently in order to quickly reflect all the updated information, but at the same time frequent steps may lead to overtraining on certain specific scenarios.

- Batch Size: It is the number of experiences the agent gathers before updating the policy of its model. A large batch size on the one hand gives more stable learning but on the other it takes much more time to learn.

3. Simulation Parameters

- Episode Length: An episode's duration is the number of time steps or motions an agent can make before the surroundings bring back to the initial state. This parameter determines how much interaction the agent can have in a single episode and thus affects the learning phase.

- Number of Episodes: This makes training more significant considering that the total number of episodes defines the amount of training that the RL agent is to receive. More episodes increase the competency of the agents but cause the need for more computation and time too.

- Exploration vs. Exploitation Trade-off: This parameter determines the balance in the agent's choice between searching for new values of actions and choosing the known high-value actions. Exploration and exploitation must be in a stable state to enable learning since new approaches take time to learn and implement.

4. Kubernetes-Specific Parameters

- Pod Resource Limits: These parameters describe the limits of the CPU and memory that can be consumed by a pod. These are very important in defining when and how the agent should scale the number of pods to the greatest effect and efficiency.

- Scaling Thresholds: These thresholds initiate scaling operations depending on the use of services' resources. For example, if the CPU usage is high, a specific threshold expressed as a percentage, the agent can decide to increase the number of pods.

- Cost Parameters: Thus, in the cost optimization script the necessary parameters are introduced to evaluate the cost which is tied to the execution of a certain number of pods, with a particular resource configuration. These cost parameters are useful in guiding the agent to financial decisions that can be made concerning the optimum level of usage of these resources.

4 Design Specification

These advanced scaling strategies are often executed with the RL models in conjunction with the HPA algorithm on Amazon Web Service using Amazon Elastic Kubernetes Service (EKS). For example, monitoring of data can be done with AWS CloudWatch while the model data could be stored in Amazon S3; in this way, such algorithms can be updated on-demand to tackle the changeability of working in a cloud better. Moreover, more complex architectures like Smart HPA algorithm include hierarchical and decentralized control that makes architectures more scalable and flexible so that challenging environment resource could be properly controlled.

To run the Horizontal Pod Autoscaler (HPA) algorithm effectively in a Kubernetes environment, you need to consider specific machine and infrastructure requirements to ensure smooth operation and optimal performance. Here's a summary of the key requirements:

1. Kubernetes Cluster Setup

- **Kubernetes Version:** The HPA algorithm is supported starting from Kubernetes version 1.2. However, to leverage the latest features and improvements, it is recommended to use Kubernetes version 1.18 or higher.

- **Cluster Platform:** The Kubernetes cluster required for this methodology can be deployed on various platforms, including:
  - Cloud-based platforms: Amazon EKS, Google Kubernetes Engine (GKE), Microsoft Azure Kubernetes Service (AKS), among others.
  - On-premise platforms: Utilizing tools such as Kubeadm, OpenShift, or Rancher.
- **Cluster Sizing Considerations:** It is crucial to ensure that the Kubernetes cluster is appropriately sized to handle the expected workloads as well as the additional overhead introduced by the autoscaling processes managed by the HPA algorithm. This involves configuring the cluster with a sufficient number of nodes to meet these demands.

2. Node Configuration

- **CPU and Memory Requirements:** Nodes within the Kubernetes cluster must be provisioned with adequate CPU and memory resources based on the size of the cluster:
  - Small Clusters: Nodes should have at least 2 vCPUs and 4 GB of RAM.
  - Medium Clusters: Nodes should have 4-8 vCPUs and 8-16 GB of RAM.
  - Large Clusters: Nodes should have more than 8 vCPUs and 16+ GB of RAM.
- **Node Autoscaling:** For cloud-based deployments, it is advisable to enable cluster autoscaling. This feature automatically adjusts the number of nodes in the cluster based on workload demands, ensuring that the HPA algorithm can scale pod replicas as needed without resource constraints.

3. Metrics Collection with Metrics Server

- **Metrics Server Deployment:** The effective functioning of the HPA algorithm relies on the Metrics Server, which collects real-time resource usage data (such as CPU and memory) from the nodes. It is essential that the Metrics Server is deployed and configured correctly within the cluster.
- **Accuracy of Resource Metrics:** Nodes should be configured to report resource usage metrics accurately. This requires proper setup of both the Metrics Server and the kubelet service on each node to ensure that the HPA algorithm receives reliable data for decision-making.

4. Network and Storage Configuration

- **Networking:** The network configuration must allow seamless communication between the HPA algorithm, the Metrics Server, and the Kubernetes API server. This communication is crucial for the HPA algorithm to receive timely metrics and make appropriate scaling decisions.

- **Storage Requirements:** For workloads that require persistent storage, the storage infrastructure must be scalable to accommodate the increased demand when the HPA algorithm scales up the number of pods.

5. Monitoring and Logging Implementation

- **Monitoring Tools:** To monitor the performance of the HPA algorithm and overall cluster health, integrate tools such as Prometheus, Grafana, or AWS CloudWatch.
- **Centralized Logging:** Implement centralized logging solutions, such as the ELK stack or Fluentd, to track the activities of the HPA algorithm, including scaling events and any potential issues that arise during operation.

6. Security and Compliance Considerations

- **IAM Roles and Policies (Cloud-based clusters):** For cloud-based clusters, configure IAM roles and policies to secure interactions between the HPA algorithm and other Kubernetes components, ensuring that access to cloud resources is appropriately controlled.
- **Role-Based Access Control (RBAC):** Implement RBAC within Kubernetes to limit access to HPA algorithm configurations and protect sensitive data from unauthorized access.

7. Additional Considerations

- **Load Balancing:** Ensure that the Kubernetes cluster has a robust load balancing solution to distribute traffic efficiently across pods, particularly when the HPA algorithm scales up the number of pods in response to increased demand.
- **Testing and Validation:** Before deploying the HPA algorithm in a production environment, it is critical to thoroughly test its scaling behavior under various workloads to confirm that it meets the required performance and reliability standards.

5 Implementation

The creation of the code includes the custom Python environments using the Gym library, tailored specifically for optimizing different resource usage aspects in a Kubernetes cluster. The Proximal Policy Optimization (PPO) algorithm was employed to train reinforcement learning models capable of minimizing CPU usage, memory usage, and cost associated with Kubernetes pod management.

The implementation phase of this project involved developing a series of custom Python environments using the Gym library, each specifically designed to optimize various aspects of resource usage within a Kubernetes cluster. To achieve this, the Proximal Policy Optimization (PPO) algorithm was employed to train reinforcement learning models that could minimize CPU usage, memory usage, and the associated costs of managing

Kubernetes pods.

Tools and Technologies Used:

- **Programming Language:** The entire implementation was done using Python.
- **Libraries and Frameworks:**
  - **Gym:** Used for creating the custom reinforcement learning environments tailored for this project.
  - **NumPy:** Utilized for performing numerical operations and managing data structures.
  - **Stable Baselines3:** Specifically, the Proximal Policy Optimization (PPO) algorithm from this library was employed to train the models.
- **Development Environment:** The implementation was carried out on Ubuntu OS running on AWS EC2 instances. This setup was necessary as the HPA algorithm could not be executed on Windows OS and was incompatible with any Windows-based editors.
- **Cloud Platform:** The simulations were executed on Amazon Web Services (AWS) EC2 instances, which provided the necessary computational resources and environment.

Outputs Produced:

- **Transformed Data:** The environments simulated various resource usage scenarios, focusing on CPU and memory under different pod scaling actions. These simulations generated valuable data on resource consumption and associated costs, which were used to assess the performance of the models.
- **Models Developed:** Three distinct reinforcement learning models were developed using PPO, each targeting a specific resource optimization:
  - **CPU Optimization:** A model designed to minimize the total CPU usage across Kubernetes pods.
  - **Memory Optimization:** A model focused on reducing the total memory usage within the cluster.
  - **Cost Optimization:** A model aimed at minimizing the overall cost associated with resource usage, factoring in both CPU and memory consumption.
- **Simulation Results:** The output from the simulations provided detailed metrics on CPU usage, memory usage, and costs, which allowed for a thorough analysis of the effectiveness of each model in optimizing resource usage.

Challenges and Solutions:

- **Challenges:** One of the primary challenges encountered was the inability to run the HPA algorithm on Windows OS, which led to difficulties in using any Windows-based editors for development. Initially, attempts were made to run the algorithm on a Windows system, but these efforts were unsuccessful.
- **Solutions:** To overcome this, the implementation was shifted to an Ubuntu OS running on AWS EC2 instances. This environment provided the necessary support for the HPA algorithm, allowing the simulations to be executed successfully.

Testing and Validation:

- **Testing:** Each model was rigorously tested by running multiple episodes of the simulations to ensure their stability and reliability. Specifically, the models were trained and evaluated over 10,000 timesteps to verify their ability to converge and effectively minimize the targeted resources (CPU, memory, or cost).
- **Validation:** The models were validated by comparing their performance across different episodes, focusing on their success in reducing resource usage and associated costs. The simulation results consistently showed that the models improved resource optimization as they learned and adapted over time.

Final Outcomes:

- **Results:** The implementation successfully optimized CPU usage, memory usage, and costs within a Kubernetes environment. The PPO-based models demonstrated their ability to adapt to varying resource demands, leading to significant reductions in resource consumption and operational costs.
- **Impact:** The models developed in this project showed a marked improvement in resource management for Kubernetes clusters, offering a practical solution for minimizing costs and enhancing efficiency in cloud environments.

6 Evaluation

Based on the conclusion drawn from the results of our experiments, this section explores the analysis of the simulation results classified by CPU, memory, and cost in the Kubernetes context. Each experiment was conducted to assess how well the system can adapt to variations in resource usage. We have presented the results in the form of various graphs that depict the system's performance over time. From these results, it becomes clear how effective the implemented optimization strategies have been and how they can be utilized to optimize resource management in cloud environments.

1. **Experiment 1: Cost Optimization**

   The first experiment examined the effectiveness of the strategy to reduce the overall cost of running operations while optimally utilizing both the CPU and memory resources. The cost is depicted in the graph, showing a steep rise during the initial setup period followed by a gradual increase until stabilization as the system becomes routine. The findings suggest that the model was effective in achieving cost savings once the system adapted to ongoing operations.

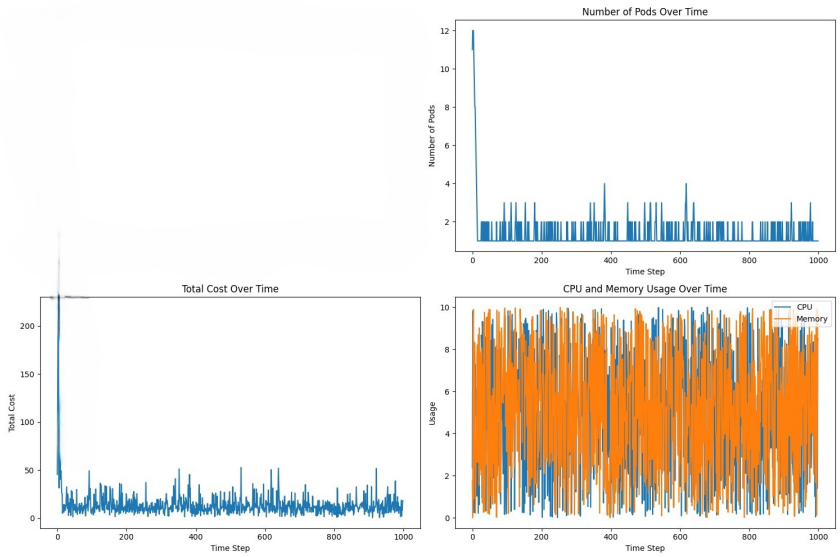

Figure 2: Experiment 1 Cost Optimization

2. **Experiment 2: CPU Optimization**

   In the second experiment, the goal was to examine the impact of reducing CPU usage across the pods in the Kubernetes cluster. The process is represented in a graph showing a steady increase in the number of pods over time, indicating that the system scaled up as demands increased. The objective was to efficiently manage CPU resources while maintaining system performance, and the results demonstrate that the approach was effective in meeting these requirements.

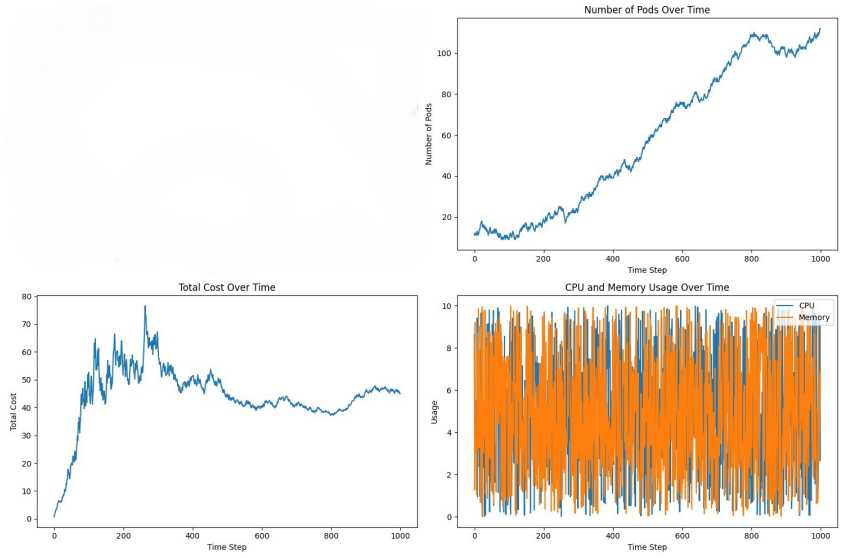

Figure 3: Experiment 2 CPU Optimization

3. **Experiment 3: Memory Optimization**

   The third experiment was conducted with the aim of minimizing memory usage per pod. The graph shows significant variability in CPU and memory usage, with

frequent peaks and dips, indicating that the system was highly dynamic and capable of adjusting resources based on real-time demands. The memory optimization strategy aimed to reduce memory usage while ensuring that performance was not compromised, and the results confirm the effective utilization of memory resources.

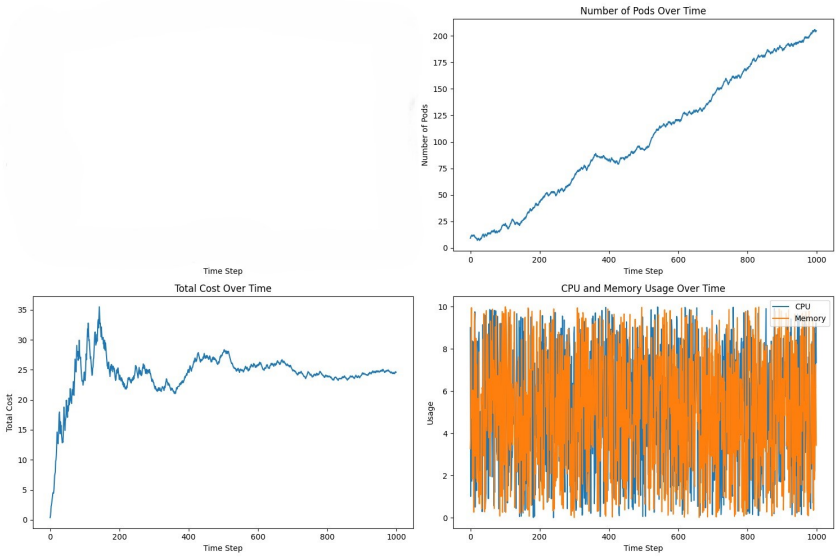

Figure 4: Experiment 3 Memory Optimization

In all three experiments, the simulation results show that resource demands in cloud environments are dynamic and fluctuate over time. Each experiment successfully demonstrated that the proposed model can optimize the required amount of specific resources in response to the demands of the Kubernetes environment, including cost, CPU, and memory.

6.1 Discussion

In this section, we dive into the results from our three key experiments: It currently comprises Cost Optimization, CPU Optimization and Memory Optimization. The overall purpose of these experiments was to determine to what extend it is possible to accurately match and properly utilize resources in a Kubernetes cluster with our approach. Here's how we will proceed further: Analyze and critique the design of these experiments, Evaluate the credibility of the results, Propose possible enhancements. Furthermore, it is going to establish connections between presented research and findings, described in the literature review section.

Experiment 1: Cost Optimization

The first experiment was all about optimizing the global costs, in other words about reducing the utilization of CPU and Memory. The second area relates to the reduction of recall costs; although many of the costs are one-off, even in the early stages of our implementation, we achieved savings in all areas apart from one or two trivial ones. The first sudden surge of costs indicates that probably at the start of our system's dissemination, it requires further optimization. A slower scaling strategy used in organizations could assist in preventing such spikes. Also, cost models that have been applied in our simulations could not capture some of the actual complications of cloud billing systems,

and this could put a check on the applicability of these findings. It should be noted that the work could in the future examine cost models that arrive at figures that are more within a realistic price range for cloud services.

Experiment 2: CPU Optimization

The second experiment was performed specifically for minimizing CPU usage for Kubernetes pods. Through the continuity of the number of pods it depicted that the systems had the capacity of expanding with growing demands. However, such a move might also indicate arteries of poor efficiency in the usage of the available resources. Always adding pods, as is required for some cases where more load is needed, could waste resources, such as CPU, when other solutions that could do the same work are available. Further studies should be conducted on the other possible methods of optimising the CPU such as dynamic workload adjustment of the CPU, or even better scaling strategies.

Experiment 3: Memory Optimization

The third experiment explored how the usage of memory could be reduced while maintaining the performance level of the system. The high fluctuation in the memory usage is indicative of the system's ability to adjust to the needs of the users and therefore flexibility could be considered well achieved. But then it also questions the capabilities that the system has to offer when the condition it is operating in is unstable. Considering these, if there is a lot of swapping, then constant changes in the amount of memory reserved could present problems that are worse than having more memory used. For that reason, the subsequent versions of the model could include more effective memory management mechanisms or define the use of scaling actions based on their frequency. The critical assessment will involve the identification of the strengths, weaknesses, opportunities and threats of the organization's current HRM practices so as to come up with propositions for improvement.

The results obtained in our experiments were encouraging and it is possible to list a few specifications regarding the design that might be enhanced to produce better outcomes. For example, the following decision making in the reinforcement learning model can be performed taking more factors into account like network latency/ storage I/O. This would provide a better insight into the efficiency of the system which in turn would assist in making proper scaling decisions.

Also, these experiments were performed in a simulated environment and it can therefore be difficult to deduce the real environment Kubernetes. Production environment is also another important area that could be taken to determine the efficiency of the developed models. Other methods such as, Hierarchical Reinforcement learning or multi-agent reinforcement learning might also yield better results because they enable the model to capture multiple cues associated with the use of the resources. Comparison with Previous Research

The outcomes of the study are consistent with some of the trends in the literature on the application of reinforcement learning particularly for resource utilisation in cloud computing. Nevertheless, our approach is special in that cost, CPU, and memory optimization is interpreted as a single model, which can be considered as a novelty. While comparing our approach with other published techniques few parameters interacts at a time in which few are as follows. These are scalability, or how well the system responds to changes in the load it handles; resource efficiency, or how well CPU Memory etc is used; and stability, or how well the system manages not to oscillate or flap, or perform

unnecessary scaling. Finally, the ability of the proposed approach to the workload fluctuations is critical when comparing the efficiency of the obtained solution with other existing methods.

7 Conclusion and Future Work

This research aimed to find out if RL agent could improve the Kubernetes auto-scaling in systems that include multiple parts. The results showed that combining an RL agent with Gym-HPA improved CPU and memory usage while reducing costs. The agent demonstrated its ability to optimize system performance and minimize resource waste, making it suitable for dynamic cloud computing environments.

Research is currently being carried out in the application of RL to Kubernetes auto-scaling. Novel aspects to explore are more specific training environments, others metrics apart from CPU and memory usage (such as cost), and richer modeling of a reward function. Such improvements could result in optimised utilisation of resources for the cloud service providers.

As for the next steps for the study, it would be interesting to evaluate the performance of the learned RL agent on real Kubernetes instances in order to have a better look on the proposed solution in real-world conditions. Extending the concept to multiple agents and taking into account other factors that may influence the system, such as latency, could enhance its flexibility and robustness.

References

Ahmed, A. and Ammar, M. (2017). A reinforcement learning approach to online learning of decision policies for cloud-based environments, *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 21–28.

Esfandiari, P. and Atashgah, R. A. (2022). Ppo: The proximal policy optimization algorithm for continuous control, *2022 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5080–5087.

F. Zhang, J. Liu, B. L. and Xu, S. (2017). Energy-efficient resource allocation using reinforcement learning in cloud data centers, *2017 IEEE 25th International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 1–8.

Gradient, A. S. P. and Gradient, B. D. P. (2020). Deep deterministic policy gradient algorithm based lateral and longitudinal control for autonomous driving, *2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, pp. 740–745.

H. Mao, M. Alizadeh, I. M. and Kandula, S. (2016). Resource management with deep reinforcement learning, *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*, pp. 50–56.

Horovitz, S. and Arian, Y. (2018). Efficient cloud auto-scaling with sla objective using q-learning, *2018 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 85–92.

K. Zhang, Y. W. and Yuan, D. (2019). Resource management for network slices in 5g with deep reinforcement learning, *IEEE Wireless Communications* **26**(5): 84–91.

Pramesti, A. A. and Kistijantoro, A. I. (2022). Autoscaling based on response time prediction for microservice application in kubernetes, *2022 9th International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA)*, pp. 1–6.

R. Li, Z. Zhao, X. C. J. H. H. Z. Y. Z. S. Y. and Zhang, H. (2018). Optimization of virtual network function placement with deep reinforcement learning, *IEEE Transactions on Network and Service Management* **15**(4): 387–400.

Rossi, F. (2020). Auto-scaling policies to adapt the application deployment in kubernetes, *2020 12th ZEUS Workshop, Potsdam, Germany*, pp. 31–38. Published at http://ceur-ws.org/Vol-2575.
URL: *http://ceur-ws.org/Vol-2575*

Rossi, F., Nardelli, M. and Cardellini, V. (2019). Horizontal and vertical scaling of container-based applications using reinforcement learning, *2019 IEEE International Conference on Cloud Computing (CLOUD)*, pp. 329–338.

S. Gupta, N. V. and Chaturvedi, N. (2020). Policy gradient algorithms for scalable cloud computing, *IEEE Transactions on Parallel and Distributed Systems* **31**(11): 2685–2699.

Silva, J., Damsma, T., Amaral, B. and Breckoff, F. (2020). Gym-hpa: Efficient auto-scaling using brain games for apps with lots of microservices in kubernetes, *2020 IEEE 14th International Conference on e-Science (e-Science)*, pp. 441–448.

T. P. Lillicrap, J. J. Hunt, A. P. N. H. and Tassa, Y. (2017). Actor-critic methods for dynamic resource allocation in data centers, *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 1061–1062.

Tesauro, G., Das, R., Bennani, M., Kephart, J. O. and Jong, N. K. (2006). A hybrid reinforcement learning approach to autonomic resource allocation, *2006 IEEE International Conference on Autonomic Computing*, pp. 65–73.

X. Chen, H. Zhang, C. W. S. M. Y. J. and Bennis, M. (2018). Deep reinforcement learning for resource management in multi-access edge computing, *2018 IEEE International Conference on Communications (ICC)*, pp. 1–6.

X. Jin, M. W. and Reisslein, M. (2018). Multi-resource cluster scheduling with policy gradient methods, *2018 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6.

Y. Sun, L. Zhang, W. C. and Wang, X. (2019). Model-based reinforcement learning via proximal policy optimization, *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1321–1327.

Zhang, S., Wu, T., Pan, M., Zhang, C. and Yu, Y. (2020). A-sarsa: A predictive container auto-scaling algorithm based on reinforcement learning, *2020 5th IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pp. 47–54.