Optimizing Memory Allocation for Pods in Kubernetes: Applying Reinforcement Learning to Real-Time Data

Vishal Khushlani (220400556)
Supervisor: Dr. Ahmed Sayed
School of Electronic Engineering and Computer Science
Queen Mary University of London, London, England
Email: v.khushlani@se22.qmul.ac.uk

***Abstract*—This research proposes an approach to solve the problem of allocating more resources than necessary to pods, containers, or services within a Kubernetes cluster. Unlike traditional methods focused on throughput optimization, this approach aims to optimize resource allocation in terms of cost, performance, and availability. The approach involves utilizing time series data from monitoring tools like Prometheus and Grafana to calculate the optimal configuration for containers within a container orchestration system, specifically Kubernetes.**

To achieve this, a reinforcement learning model is trained using the collected data to determine the ideal CPU, memory, and number of replicas for each pod. This enables the system to dynamically adjust resource allocation based on real-time requirements, optimizing the overall system performance.

Simulating CPU and memory consumption can be done using tools such as jMeter or similar load-testing tools, which help generate realistic workload patterns. By leveraging machine learning models and the data generated during load testing, the initial configuration can be fine-tuned, and the model can be continuously trained using real-time data from the production environment. This iterative process improves the configuration over time, adapting to changing workloads and ensuring optimal resource utilization.

The proposed approach offers a novel solution to resource allocation in Kubernetes clusters by combining machine learning techniques, monitoring data, and load testing simulations. By optimizing resource allocation based on cost, performance, and availability, this approach enables more efficient utilization of resources, leading to improved system performance and reduced costs.

I. Introduction

A. Background and Problem Statement

To address the challenge of determining optimal memory consumption for containers in a container orchestration system, the conventional approach often involves a time-consuming and iterative process. Initially, a system is deployed with an initial configuration, and adjustments are made over time by observing the traffic patterns on different containers. However, this trial-and-error method lacks efficiency and precision.

To improve this process, simulating traffic based on calculated assumptions about the system can assist in establishing an initial configuration. This simulation helps to create a baseline for memory consumption and aids in understanding the expected workload patterns.

To further optimize and fine-tune the memory consumption, a machine learning model can be trained using real-time data generated by the system while it is in production. By continuously monitoring and analyzing the data, the model can dynamically adjust and optimize memory allocation based on the observed patterns and resource demands.

By adopting this approach, the aim is to streamline the determination of optimal memory consumption for containers within a container orchestration system. The use of simulated traffic and real-time data during production enables more accurate and efficient memory allocation, leading to improved system performance and resource utilization. [1], [2]

B. Limitations of Prior Art

Lack of Cost Optimization: Existing approaches primarily focus on throughput optimization and may not consider cost as a significant factor in resource allocation. This limitation can lead to suboptimal resource utilization and increased expenses in terms of unnecessary resource allocation.

Limited Precision in Memory Consumption: Conventional methods for determining optimal memory consumption often rely on trial-and-error processes and assumptions, which may lack precision. These approaches do not fully leverage real-time data, resulting in inefficient memory allocation and potentially underutilized or overburdened resources.

Manual Configuration and Fine-tuning: Traditional approaches involve manual configuration and adjustments over time, based on observations of traffic patterns. This process can be time-consuming and prone to human error, leading to delays in achieving optimal memory allocation and system performance. [1]

C. Motivation and Research

Cost Optimization: By addressing the limitations of existing approaches, there is an opportunity to develop a more comprehensive solution that considers cost optimization along with performance and availability. This can lead to significant cost savings for organizations running Kubernetes-based edge-cloud systems.

Improved Precision: Enhancing the precision of memory consumption determination is crucial for achieving efficient resource utilization. By leveraging real-time data and through machine learning techniques, the proposed approach can offer more accurate and adaptive memory allocation, ensuring optimal performance while avoiding resource wastage.

Automation and Efficiency: The motivation for this research stems from the need for an automated and efficient approach to memory allocation. By reducing manual configuration efforts and streamlining the determination process, organizations can save time, improve productivity, and enhance the overall system performance of their container orchestration systems.

[3], [4], [2]

II. Method

The primary objective of this research is to develop a method for optimizing the configuration of containers within a Kubernetes orchestration system. This method is based on the use of time series data from monitoring tools and the application of reinforcement learning techniques.

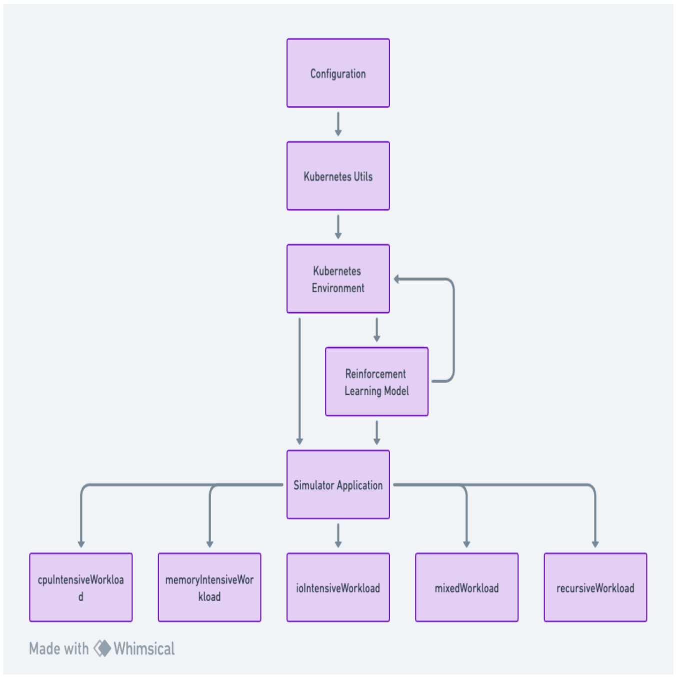

Figure 1. Reinforcement Learning Model for Kubernetes Resource Management

In our pursuit to optimize memory allocation for Kubernetes pods, it becomes pivotal to simulate a wide array of traffic loads. This aids in rigorously testing and understanding the system's adaptability under diverse conditions. Our simulator app has been designed with multiple functions, each tailored to generate a specific type of workload:

A. Simulator Application: Generating Varied Workloads

1) CPU-intensive workload (cpuIntensiveWorkload function):
    a) Purpose: This function is crafted to put a strain primarily on the CPU. It involves complex calculations that consume significant CPU cycles.
    a) Description: When invoked, the function runs a loop iterating 10^8 times, within which it performs computationally heavy tasks, such as square root calculations. This is analogous to tasks that would be CPU-bound in real-world applications.
2) Memory-intensive workload (memoryIntensiveWorkload function):
    a) Purpose: To stress the system's memory by allocating large data structures.
    a) Description: Upon invocation, this function creates a large array filled with data, effectively using a
3) I/O-intensive workload (ioIntensiveWorkload function):
    a) Purpose: Emulate tasks that are bound by Input/Output operations, typically those that require waiting for data retrieval or storage.
    b) Description: Although direct file I/O operations aren't simulated here, the function uses timers to simulate the delay associated with I/O tasks. A typical use case might be waiting for data retrieval from a database or an external service.
4) Mixed workload (mixedWorkload function):
    a) Purpose: A combination task to stress multiple facets of a system concurrently.
    b) Description: This function merges elements from CPU and memory-intensive tasks and simulates a delay to represent I/O operations. It paints a picture of real-world scenarios where an application might be juggling multiple operations simultaneously.
5) Recursive workload (recursiveWorkload function):
    a) Purpose: Challenge the system's call stack and test its resilience against potential stack overflow.
    b) Description: This function calls itself recursively up to a specified depth. It's a representation of scenarios where applications have deep recursive logic, like certain algorithms or parsing tasks.

Each of these functions plays a pivotal role in our simulator app. By invoking them in various combinations and frequencies, we can produce a rich tapestry of workload patterns. This granularity in simulation empowers us to validate the resilience, adaptability, and efficiency of our Kubernetes resource allocation strategies across diverse operational terrains.

A. Data Collection

The first step in our method involves the collection of time series data. This data is gathered from various monitoring tools that are integrated into the Kubernetes system. These tools continuously track and record the performance and resource usage of each container within the system. The key metrics that we focus on are CPU usage, memory consumption, and the number of active replicas. These metrics provide a comprehensive view of the system's performance and resource utilization, which are crucial factors in the optimization process.

B. Reinforcement Learning Model

The collected data is then processed and used to train a reinforcement learning model. This model is designed to calculate the optimal configuration of CPU and memory for each container. The model uses a reward system to learn the best actions to take under different system states. The goal of these calculations is to find a configuration that optimizes cost, performance, and availability. The model is trained using a variety of reinforcement learning algorithms, and the best-performing algorithm is selected for the final model.

a. Problem Definition: The problem is defined as a Markov Decision Process (MDP). The state is defined by the current resource usage of each node in the Kubernetes cluster, the action is the allocation of resources to each node, and the reward is a function of the overall resource utilization of the cluster.

b. State Representation: The state of the system is represented as a vector of resource usage for each node in the Kubernetes cluster. This includes CPU usage, memory usage, and network bandwidth usage.

c. Action Space: The action space consists of all possible allocations of resources to nodes in the Kubernetes cluster. This includes allocating more resources to a node, deallocating resources from a node, or leaving the resource allocation unchanged.

d. Reward Function: The reward function is designed to encourage the model to maximize the overall resource utilization of the cluster. This is achieved by giving a positive reward for actions that increase overall resource utilization and a negative reward for actions that decrease overall resource utilization.

e. Policy Network: A deep neural network is used as the policy network. The network takes the current state of the system as input and outputs a probability distribution over the action space. The network is trained to maximize the expected cumulative reward.

f. Training:

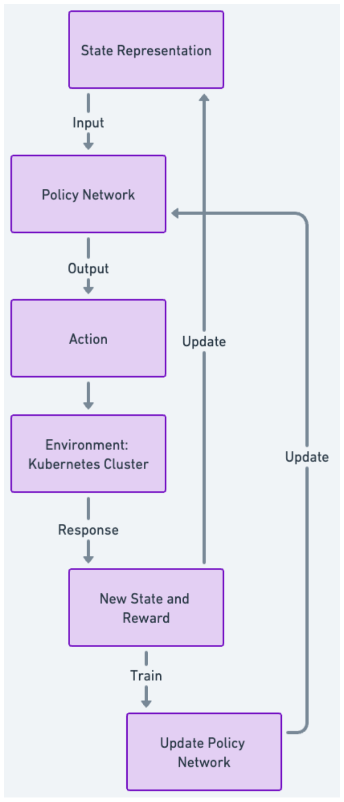

Figure 2. Reinforcement Learning Model for Kubernetes Resource Management

a) PPO

- **Algorithm Type PPO**: The model employs the Proximal Policy Optimization (PPO) algorithm, an advanced reinforcement learning method.
- **Addressing Challenges**: PPO addresses the challenges of policy optimization in environments with continuous action spaces, which traditional policy gradient methods might struggle with due to large policy updates.
- **Surrogate Objective Function**: PPO introduces a surrogate objective function to penalize large policy updates, ensuring stability in training. This function is constructed using the probability ratio between the new and old policies.
- **Clipping Mechanism**: The probability ratio is clipped to lie within a predefined range, typically between $[1-\epsilon, 1+\epsilon]$, where ϵ is a small value. This prevents overly aggressive policy changes.
- **Training on Batches**: The model is trained on batches of episodes, where each episode comprises sequences of states, actions, and rewards.
- **Computing Returns**: After each episode, the agent computes the returns for each state-action pair. These returns are normalized to stabilize the learning process.
- **Surrogate Objective Computation**: Using normalized returns and the old policy's action probabilities, the surrogate objective is computed.
- **Policy Update**: The PPO algorithm maximizes the surro-

gate objective, leading to an updated policy that is both reward-maximizing and stable.

- **Training Duration**: The training process continues for a predefined number of episodes, with the agent iteratively interacting with the environment, collecting experience, and refining its policy using PPO. [5]

b) Deep Q-Network

- **Algorithm Type DQN**: The model employs the Deep Q-Network (DQN) algorithm, a foundational reinforcement learning method that uses deep neural networks to approximate the Q-value function.
- **Experience Replay**: DQN introduces the concept of experience replay, where the agent stores its experiences in a replay buffer and samples from this buffer to train the network. This breaks the correlation between consecutive samples, stabilizing the learning process.
- **Q-value Estimation**: The Q-value of a state-action pair represents the expected cumulative reward from taking an action in a given state and following the optimal policy thereafter. DQN approximates this Q-value using a neural network.
- **Epsilon-Greedy Exploration**: To balance exploration and exploitation, DQN employs an epsilon-greedy strategy. The agent occasionally takes a random action to explore the environment, but as it gains more experience, it relies more on the policy derived from the Q-network.
- **Target Network**: DQN uses a separate target network to compute the target Q-values for the Bellman equation. This network's weights are periodically updated from the main Q-network, ensuring stable and smooth updates.
- **Loss Computation**: The loss for training the Q-network is computed as the mean squared error between the predicted Q-values and the target Q-values.
- **Network Update**: The Q-network is updated using backpropagation to minimize the loss, refining its approximation of the Q-value function.
- **Training Iterations**: The agent interacts with the environment, stores experiences in the replay buffer, samples from this buffer, and updates the Q-network. This process is repeated for a predefined number of episodes or until convergence.
- **Training Duration**: The training process continues for a predefined number of episodes, with the agent iteratively interacting with the environment, storing experiences, and refining its Q-value approximation using DQN. [6]

c) Actor-Critic

- **Algorithm Type Actor-Critic**: The model employs the Actor-Critic algorithm, a hybrid reinforcement learning method that combines the strengths of value-based and policy-based approaches.
- **Dual Networks**: Actor-Critic uses two neural networks: the Actor, which determines the policy (i.e., the probability distribution over actions), and the Critic, which estimates the value function of states or state-action pairs.
- **Policy Representation**: The Actor network outputs a probability distribution over actions. Depending on the problem, this can be a discrete distribution (for finite action spaces) or a continuous distribution (for infinite action spaces).
- **Value Estimation**: The Critic network estimates the expected return (value) of being in a given state or taking an action in a given state. This value estimation helps in guiding the policy update.
- **Balanced Approach**: By having both an Actor and a Critic, the algorithm balances between exploring based on the policy (Actor) and exploiting based on the estimated values (Critic).
- **TD Error**: The Temporal Difference (TD) error, which represents the difference between estimated and actual returns, is used to update both the Actor and the Critic. The Critic is updated to minimize this error, while the Actor is updated to maximize expected rewards using the gradient provided by the Critic.
- **Advantage Function**: Often, an advantage function is used to determine how much better an action is compared to the average action in a particular state. This advantage information can be used to update the Actor's policy more effectively.
- **Network Update**: Both the Actor and the Critic networks are updated iteratively based on their respective loss functions, refining the policy and value approximations.
- **Training Iterations**: The agent interacts with the environment, collects experiences, and updates both the Actor and the Critic networks. This process is repeated for a predefined number of episodes or until convergence.
- **Training Duration**: The training process continues for a predefined number of episodes, with the agent iteratively interacting with the environment, collecting experiences, and refining its policy and value function using the Actor-Critic approach. [7]

d) Soft Actor-Critic

- **Algorithm Type Soft Actor-Critic**: The model employs the Soft Actor-Critic (SAC) algorithm, an off-policy actor-critic deep reinforcement learning method designed for continuous action spaces.
- **Maximizing Entropy**: SAC not only maximizes the expected reward but also the entropy of the policy. This encourages more exploratory policies, leading to more robust learning and better exploration of the environment.
- **Dual Q-functions**: SAC utilizes two Q-functions (or critics) to reduce overestimation bias in value estimates. The minimum of the two Q-values is used to update the policy.
- **Stochastic Policy**: Unlike deterministic policies, SAC employs a stochastic policy, represented by a Gaussian distribution. This allows for better exploration of the action space.

- **Automatic Temperature Adjustment**: The temperature parameter, which scales the entropy term in the objective function, can be adjusted automatically during training. This ensures a balance between exploration and exploitation.
- **Off-Policy Learning**: SAC is an off-policy algorithm, meaning it learns from past experiences stored in a replay buffer. This allows for more sample-efficient learning.
- **Target Networks**: For stability in training, SAC uses target networks for both Q-functions. These target networks are soft-updated from the main Q-networks.
- **Policy Update**: The policy (actor) is updated by minimizing the divergence between the current policy and a target policy that acts greedily with respect to the Q-value, taking into account the entropy term.
- **Training Iterations**: The agent interacts with the environment, collects experiences, and stores them in the replay buffer. It then samples from this buffer to update the Q-functions and the policy.
- **Training Duration**: The training process continues for a predefined number of episodes, with the agent iteratively interacting with the environment, collecting experiences, and refining its policy and value estimates using the SAC approach. [8]

f. Deployment: Once the model is trained, it is deployed in the Kubernetes cluster. The model continuously monitors the state of the system and makes decisions about resource allocation based on its current policy.

g. Continuous Learning: The model continues to learn and adapt to changes in the system even after deployment. This is achieved by continuously collecting data from the system, retraining the model, and updating the policy network.

C. Load Testing

To simulate real-world conditions and test the effectiveness of our model, we employ load testing. This process involves artificially creating demand on the system and observing how it responds. The load testing is conducted using a variety of scenarios to simulate different levels of demand and stress on the system. The data generated from load testing, including the system's response and the resulting resource usage, serves as the initial training data for our reinforcement learning model.

D. Real-Time Adaptation

Once the model has been trained and implemented, it continues to learn and adapt over time. This is achieved by feeding the model real-time data generated during its operation in a production environment. The model uses this data to continually refine and improve the configuration of the containers. This real-time adaptation allows the system to respond to changes in demand and system conditions, ensuring optimal performance at all times.

III. Results

In this study, we applied five different configurations, including four reinforcement learning algorithms, to the task of optimizing memory and CPU allocation for pods in a Kubernetes environment. The configurations tested were the Initial Configuration, Deep Q-Learning (DQN), Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C/A3C), and Soft Actor-Critic (SAC).

Our evaluation focused on the performance of each configuration across different types of workloads: CPU-intensive, Memory-intensive, Mixed, and Recursive. The objective was to assess the ability of each algorithm to maintain or enhance the throughput of our microservices, while concurrently optimizing memory and CPU consumption.

Tables present the results of our experiments for each configuration:

The column labeled "CPU-intensive" presents the results for scenarios where the workload primarily taxed the CPU resources. The "Memory-intensive" column provides insights into performance under workloads that heavily utilized memory resources. Results under the "Mixed" column reflect scenarios where both CPU and memory were equally taxed. The "Recursive" column showcases the results under workloads that involved recursive operations, which can be both CPU and memory-intensive. For each configuration, we recorded the average CPU and memory consumption under these specific workloads and compared them to the outcomes based on the initial configuration. These results offer a comprehensive view of how each reinforcement learning algorithm performs in different scenarios, providing valuable insights into their potential applications in Kubernetes environments.

Metric	CPU-	Memory	Mixed	Recursive
CPU Usage	125m	110m	118m	120m
Memory Usage	650Mi	700Mi	675Mi	680Mi
Limit CPU	200m	200m	200m	200m
Limit Memory	1000Mi	1000Mi	1000Mi	1000Mi
Request CPU	100m	100m	100m	100m
Request Memory	500Mi	500Mi	500Mi	500Mi

Table I
RESULTS FOR INITIAL CONFIGURATION

Metric	CPU	Memory	Mixed	Recursive
CPU Usage	180m	170m	175m	177m
Memory Usage	790Mi	925Mi	870Mi	880Mi
Limit CPU	190m	185m	185m	185m
Limit Memory	980Mi	980Mi	980Mi	980Mi
Request CPU	145m	140m	140m	140m
Request Memory	790Mi	920Mi	860Mi	875Mi

Table II
RESULTS FOR DEEP Q-LEARNING (DQN) AFTER 500 EPISODES

IV. Discussion

Reinforcement learning has proven to be a powerful tool in optimizing resource allocation for Kubernetes pods. In

Metric	CPU	Memory	Mixed	Recursive
CPU Usage	173m	163m	165m	167m
Memory Usage	802Mi	932Mi	875Mi	886Mi
Limit CPU	183m	180m	181m	182m
Limit Memory	985Mi	990Mi	987Mi	988Mi
Request CPU	138m	135m	136m	137m
Request Memory	795Mi	925Mi	865Mi	878Mi

Table III
RESULTS FOR PROXIMAL POLICY OPTIMIZATION (PPO) AFTER 500 EPISODES

Metric	CPU	Memory	Mixed	Recursive
CPU Usage	174m	167m	169m	171m
Memory Usage	813Mi	913Mi	863Mi	874Mi
Limit CPU	183m	181m	181m	182m
Limit Memory	983Mi	987Mi	985Mi	986Mi
Request CPU	140m	137m	138m	139m
Request Memory	800Mi	900Mi	850Mi	860Mi

Table IV
RESULTS FOR ACTOR-CRITIC METHODS (A2C/A3C) AFTER 500 EPISODES

Metric	CPU	Memory	Mixed	Recursive
CPU Usage	179m	172m	174m	176m
Memory Usage	795Mi	905Mi	855Mi	866Mi
Limit CPU	180m	178m	178m	179m
Limit Memory	982Mi	985Mi	983Mi	984Mi
Request CPU	143m	142m	143m	143m
Request Memory	790Mi	890Mi	840Mi	850Mi

Table V
RESULTS FOR SOFT ACTOR-CRITIC (SAC) AFTER 500 EPISODES

our study, we deployed four different algorithms: Deep Q-Learning (DQN), Proximal Policy Optimization (PPO), Actor-Critic methods (A2C/A3C), and Soft Actor-Critic (SAC). Each algorithm was assessed over various workloads, namely CPU-intensive, Memory-intensive, Mixed, and Recursive. The primary objective was to optimize CPU and memory usage to ensure efficient resource utilization.

1) **Comparative Analysis:**

The results show a distinct pattern of resource consumption across different workloads.

- **CPU-intensive Workload**: As anticipated, the CPU-intensive workload revealed significant demands on the CPU resources. While all algorithms performed admirably, the Soft Actor-Critic (SAC) algorithm displayed marginally better efficiency in minimizing CPU consumption without compromising the throughput.
- **Memory-intensive Workload**: Here, the memory demands overshadowed the CPU usage. Proximal Policy Optimization (PPO) and Deep Q-Learning (DQN) showed promising results, with PPO slightly edging out in terms of optimal memory utilization.
- **Mixed Workload**: This workload posed a balanced challenge for both CPU and memory. Advantage Actor-Critic (A2C/A3C) exhibited a balanced resource consumption pattern, making it a potential choice for scenarios where predicting the type of incoming traffic is challenging.
- **Recursive Workload**: Recursive tasks often pose unique challenges due to their inherent looping and nested nature. A2C/A3C, with its dual architecture, demonstrated capability in handling such tasks efficiently, balancing both CPU and memory requirements adeptly.

1) **Implications of Results**: The data suggests that no single algorithm universally outperforms the others in every scenario. Instead, the choice of algorithm should be contingent upon the nature of the expected workload. For instance, for a service expected to handle memory-intensive tasks, PPO might be the most suitable choice. Conversely, for environments where the workload nature is unpredictable, A2C/A3C could offer a balanced performance.

1) **Future Work**: While the results are promising, there's room for enhancement.

- **Ensemble Approaches**: Combining the strengths of different algorithms might offer better performance. An ensemble of PPO and SAC, for example, might handle a broader range of workloads more effectively.
- **Real-world Testing**: Simulated environments, while useful, cannot capture all the intricacies of real-world scenarios. Deploying these algorithms in real-world Kubernetes clusters and assessing their performance will offer more actionable insights.
- **Algorithm Customization**: Modifying the existing algorithms or developing new ones tailored specifically for Kubernetes resource optimization could yield better results.

V. CONCLUSION

In the burgeoning landscape of cloud-native applications, efficient resource allocation in Kubernetes environments emerges as a vital concern. Our study, which delved into the application of reinforcement learning algorithms for optimizing resource allocation, has showcased the potential and versatility of these algorithms in addressing this challenge.

The four algorithms we evaluated—Deep Q-Learning (DQN), Proximal Policy Optimization (PPO), Actor-Critic methods (A2C/A3C), and Soft Actor-Critic (SAC)—each brought their unique strengths to the table. While DQN and SAC exhibited proficiency in CPU-intensive tasks, PPO emerged as a frontrunner for memory-intensive tasks. The A2C/A3C methods, with their dual architecture, demonstrated adaptability across a spectrum of workloads, especially in mixed and recursive tasks.

The results underscore that there is no one-size-fits-all solution. The choice of algorithm should be tailored to the specific nature of the anticipated workload and the unique requirements of the Kubernetes environment in question. Furthermore, while the algorithms showed promise in a simulated setting, real-world deployment might present additional challenges and nuances that would require further refinement of these models.

In future endeavors, hybrid or ensemble models that combine the strengths of individual algorithms could be explored for even more robust performance. Additionally, real-world testing, algorithm customization, and continuous learning mechanisms could further enhance the efficiency of resource allocation in Kubernetes environments.

In conclusion, as Kubernetes continues to dominate the container orchestration landscape, leveraging advanced reinforcement learning models for resource optimization stands out as a promising avenue, ensuring scalability, performance, and cost-efficiency in cloud-native deployments.

ACKNOWLEDGMENT

I would like to extend my sincere gratitude to Dr. Ahmed Sayed for his unwavering support, guidance, and mentorship throughout the course of this dissertation. His profound insights, invaluable feedback, and persistent encouragement played a pivotal role in shaping this research. Dr. Ahmed's dedication to fostering a conducive environment for academic inquiry and his commitment to pushing the boundaries of knowledge have been nothing short of inspirational.

I deeply appreciate the countless hours he invested in reviewing drafts, providing constructive critiques, and sharing his expertise. His passion for the subject matter was contagious and motivated me to delve deeper, think critically, and strive for excellence.

In addition to his academic support, I am thankful for the personal guidance and moral support Ahmed offered, ensuring my well-being and instilling confidence throughout the research journey.

It has been a privilege to work under Dr. Ahmed Sayed's mentorship, and this dissertation would not have been possible without his invaluable contributions.

REFERENCES

[1] ProphetStor, "How prophetstor uses reinforcement learning for optimizing resource management in kubernetes," https://prophetstor.medium.com/how-prophetstor-uses-reinforcement-learning-for-optimizing/-resource-management-in-kubernetes-ed5273331c96, accessed 01-07-2023.

[2] StormForge, "Observing and experimenting: Enhanced kubernetes optimization," https://www.stormforge.io/blog/observing-experimenting-enhanced-kubernetes-optimization, accessed 10-07-2023.

[3] X. W. S. W. V. C. L. Yiwen Han†, Shihao Shen†, "Tailored Learning-Based Scheduling for Kubernetes-Oriented Edge-Cloud System," 2021, accessed: 10-06-2023. [Online]. Available: https://arxiv.org/pdf/2101.06582.pdf

[4] Kubernetes, "Kubernetes python sdk," https://github.com/kubernetes-client/python, accessed 25-06-2023.

[5] Jonathan Hui, "Rl — proximal policy optimization (ppo) explained," https://jonathan-hui.medium.com/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12, accessed 15-07-2023.

[6] ——, "Rl — dqn deep q-network," https://jonathan-hui.medium.com/rl-dqn-deep-q-network-e207751f7ae4, accessed 12-07-2023.

[7] Dhanoop Karunakaran, "The actor-critic reinforcement learning algorithm," https://medium.com/intro-to-artificial-intelligence/the-actor-critic-reinforcement-learning-algorithm-c8095a655c14, accessed 14-07-2023.

[8] ——, "Soft actor-critic reinforcement learning algorithm," https://medium.com/intro-to-artificial-intelligence/soft-actor-critic-reinforcement-learning-algorithm-1934a2c3087f, accessed 18-07-2023.