National College of Ireland

Reinforcement Learning-Based Cost and Latency Optimization in Kubernetes-Enabled Multi-Cloud Environments

MSc Research Project

Master of Science in Cloud Computing

Sai Kumar Reddy Palnati

Student ID: 24104191

School of Computing

National College of Ireland

Supervisor: Yasantha Samarawickrama

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Sai Kumar Reddy Palnati
Student ID:	24104191
Programme:	MSC in Cloud Computing **Year:** 2025-2026
Module:	Research Project
Supervisor:	Yasantha Samarawickrama
Submission Due Date:	28th January 2026
Project Title:	Reinforcement Learning-Based Cost and Latency Optimization in Kubernetes-Enabled Multi-Cloud Environments
Word Count:	4,921 **Page Count:** 18

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: Sai Kumar Reddy Palnati

Date: 28th January 2026

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):	

Reinforcement Learning-Based Cost and Latency Optimization in Kubernetes-Enabled Multi-Cloud Environments

Sai Kumar Reddy Palanti

24104191

Abstract

In this project, I developed a reinforcement learning-based scheduler for Kubernetes that can smartly figure out where to place new pods, whether on a simulated AWS cluster or an Azure one—to keep both costs and response times as low as possible. Because of some organizational rules, I wasn't able to spin up real AWS EKS or Azure AKS clusters, so instead I used Kind (that's Kubernetes running inside Docker containers) to create two totally separate local clusters on my laptop, one standing in for AWS and the other for Azure. To make everything as realistic as possible, I pulled the latest on-demand pricing straight from the official AWS and Azure websites for December 2025 and used real-world latency numbers between regions that I found there too. I have developed a custom environment by using Gymnasium, where it is a popular toolkit for RL agents training purposes. I have used Locust for managing workloads, where it is an open-source tool for testing artificial work loads and to simulate real application traffic. I have trained the PPO agent by using RLlib, and I have done all the training and testing in my MacBook Air with the M2 chip—no servers needed. To train the agent, it took approximately 80 iterations, but when the training was done, the agent was able to hit an average cost of around $1.23, but the kubernetes default scheduler was hitting around $4.77 in the same work load. This can tell us a good drop of 74% in cost and also improvement in latency of about 7 to 10% due to avoiding overloading by the agent in the cluster. The best thing in my project is everything is reproducible, anyone is able to run on their own machine and I have put the entire code in my Github with clear instructions of the configuration manual where we can configure and set up Kind clusters and running Locust. I'm excited to see where this could go if we eventually move it to real cloud environments.

1 Introduction

Now-a-days, all the companies are putting a lot of money into their cloud services, where each year in recent times it is growing way faster, where global spending on public cloud services is hitting around $723 billion in 2025 alone. In the present situation all the organisations were spreading their apps data to multiple cloud providers, not just sticking to one cloud provider alone. Where this multi-cloud will help for the organisations to avoid vendor lockin and makes the system more reliable. But the standard kubernetes scheduler, which is popular for managing containerized apps, really only looks at basic things like how much CPU or memory a node has left. It has no clue about the real costs, like hourly rates for different instance types in AWS versus Azure, or how much delay there might be when traffic has to jump between regions in different clouds. That means pods, which are using little units of running apps, will often end up on the pricier or slower options, even when there's a better, cheaper spot available just across the cloud.

Originally, I had planned to set everything up on actual AWS EKS and Azure AKS clusters to test this in the real world. But the organization's accounts for those services have strict restrictions, especially for student projects, so creating real clusters wasn't allowed. After talking it over with my supervisor, we decided the best way forward was to set up a multi-cloud setup right on my local machine using Kind, which lets you run full Kubernetes clusters inside Docker containers. I made two completely separate ones—one acting as the AWS side and one as Azure—and fed in actual on-demand pricing and latency figures pulled from the vendors' public websites for late 2025. This kept things realistic without running up any cloud bills.

1.1 Background

For orchestrating containers in production, Kubernetes has become more popular, where it handles everything which includes scaling apps and also running them smoothly. Where the kubernetes default scheduler works well for allocating basic resources, but now-a-days all the companies are going towards multi-cloud for flexibility and for smarter decisions. Where, varying hourly cost can lead to waste of money and slower performance, especially as cloud bills keep increasing more and teams look for ways to optimize without locking into one cloud provider.

1.2 Research Question

Can a reinforcement learning agent, trained in a simulated multi-cloud Kubernetes environment using only publicly available pricing and latency data, and learning to schedule pods in a way that meaningfully cuts costs and improves response times compared to the standard Kubernetes scheduler?

1.3 Research Objectives

The main goal is to build and test a system that could make smarter placement. Specifically, I wanted to create a custom Gymnasium environment that pulled together real vendor pricing and inter-region latencies, and live metrics from Prometheus to make training realistic. Training a PPO agent from here to mostly pick the lower-cost options while keeping checking the performance of the system. I also built a real-time scheduler extender using Python and Flask that fetches into Kubernetes and queries to train models for decisions. Finally, testing to show clear improvements, like the 74% drop in costs we ended up seeing.

Now we have solid proof that combines into something practical, with a trained agent that outperforms the default scheduler, a working integration into Kubernetes, and evaluations that highlight the potential savings for organisation, even in a local simulation. It's a strong starting point that shows how an AI can step in where traditional schedulers leave money on the table.

2 Related Work

When we are working on scheduling in Kubernetes by using techniques like reinforcement learning for managing cloud resources, there is much research already existing, but most of it stays within the boundaries of a single cloud provider or misses some key real-world factors. My project builds on these ideas while trying to push into an area that hasn't been explored as much. Especially combining actual vendor pricing with multi-cloud decisions in a practical example.

2.1 Traditional Kubernetes Scheduling

The default Kubernetes scheduler will be reliable and does the job for most of the basic needs from many years. This Kubernetes scheduler will be working on a set of rules to check and filter nodes which can’t run a pod whenever there is not enough CPU or memory for the further processing. This scheduler will work on a priority basis to pick the best one on the remaining nodes to give the best output. It will work well for keeping the resources balanced in a single cluster. But by the time, organisations have added helpful tools like the Descheduler, which can move pods to fix poor initial placements, which is great for quickly adding or removing nodes to pack things efficiently and save money on autoscaling. But all these features are not really helpful in organisations to set up a multi-cloud environment. They have no awareness of how much each node is actually costing per hour in different providers, or the extra delay that comes from network latency when pods in one cloud need to talk to services in another. So, the organisations can end up paying more than necessary or putting up with slower performance just because the scheduler can't know all these differences in real scenarios.

2.2 Reinforcement Learning in Cloud Resource Management

Reinforcement learning has become one of the most researched algorithms for cloud and datacenter management, especially where traditional rules are unable to take the complex decisions. One of the early works was DeepRM back in 2016 by Mao and colleagues, which used RL to schedule the jobs efficiently onto servers, kind of like a smarter version of the bin-packing problem, and it showed improvements over basic methods. But in recent years in 2022, a team in Google worked on a reinforcement learning system they have designed scheduling tasks across their large datacenters, where it handled real-world scale but wasn't built specifically for Kubernetes. In 2023, RL-Kube from Zhang and others applied a DQN algorithm directly to scheduling in a single Google Kubernetes Engine cluster and got some good results on resource utilization.

But when you look for work in multi-cloud environments there is very little research. There aren't many papers which tackle the challenge of spreading workloads across different providers while balancing cost and performance. The very near one I found was a 2024 paper titled "Multi-Cloud Kubernetes Cost Optimization using Deep Reinforcement Learning," which used a different RL approach called DDPG to make decisions. Another recent year paper from 2025, where it focused on "Latency-Aware Scheduling for Edge-Cloud Continuum" did a good job prioritizing low response times, but it was not focused on cost on the other side.

We have found different papers in recent years but none of them can help us to prioritize both cost and latency based scheduling on multi-cloud environments. So, now I am going to use the stats up to 2025 from getting pricing from both AWS and Azure official sites. Getting the real time performance metrics from Prometheus during training, and doing it all in a local simulation that mimics two separate clouds without needing real accounts. This is the main gap between my work and already existing work. This set up can be easily cloned to any one and can see the live performance without using real cloud service. So, this way they can save money for not paying to cloud providers while simulating the jobs.

3 Research Methodology

For this project, I have worked and built things step by step where they were tested early and fixed which didn't work. This iterative approach was helpful because the whole setup involved quite a few moving parts like Kubernetes clusters, reinforcement learning training, real-time metrics, and integration with the scheduler. So, it was better to get small pieces working first before putting everything together. I have used my local system for all the development and testing which has an M2 chip, which is powerful enough for everything, including training the agent locally without needing any external cloud GPUs for the training.

3.1 Development Environment Setup

The first thing we had to do was to get our local system for development and testing and install all the required tools step by step. I started with Homebrew to install the basics, then set up Miniconda so that we can manage different Python environments easily and keep dependencies without overlapping. Docker Desktop was next because Kind runs Kubernetes clusters inside Docker containers, and I also needed it for pulling images and running Prometheus. I allocated all the resources to docker where giving it more memory and CPU, because of this in early we can run some slowdowns when both clusters were running at the same time.

3.2 Creating the Local Multi-Cloud Simulation

We have to create a real multi-cloud environment without actually spending money or breaking any rules. Here I have used Kind to create a separate Kubernetes cluster on my machine. I have named them "kind-aws" and the other "kind-azure" just to keep things clear in my head. Each cluster had its own control plane and worker nodes to differentiate both clusters. I have used slightly different labels so the scheduler could easily tell them. I also chose different node sizes to resemble typical AWS and Azure instance types. To make the setup more realistic, I created a basic overlay network between the clusters. I have diverted all the traffic through a small proxy script that added the latency values, which I had collected earlier. This helped simulate the slower communication you usually see between clusters running in different regions.

3.3 Metrics Collection with Prometheus

Here we have to keep track of clusters and what is happening inside them, because this was very important to train agent. I have installed Prometheus on both the kind-aws and kind-azure clusters using the official Helm chart from the official Prometheus organisation. I have taken the standard Kubernetes metrics like CPU and memory usage per node, pod status, and network traffic while developing. During training and testing, the Gymnasium environment would query Prometheus from both clusters in real time to get the current load in the environment, where this helped the agent see when one cluster was getting busy and might cause latency spikes if more pods were added there. Having the live metrics have helped the scheduler to take the decisions easily.

3.4 Gathering Real-World Pricing and Latency Data

To get all the real pricing and latency data, In December 2025, I went to the official AWS and Azure sites to get all the pricing and latest on-demand rates for instance types—like general-purpose VMs in similar regions. Here I choosed to pick the numbers that were close in performance but different in price where the agent could learn from (Azure came out cheaper for the types I choose, which is often the case). For latency, I used publicly available data from AWS's and Azure's inter-region latency tables and tools like CloudPing and Azure Speed Test to get average round-trip times between, say, US East and West Europe regions.

3.5 Building the Custom Gymnasium Environment

The main part of the training setup was a custom environment I wrote from scratch which is k8s_multi_cloud_env.py. It has all the features from Gymnasium's base class and also simulated the arrival of new pods, kept track of where everything was placed, calculated

running costs based on time and pricing which took in before, and added latency penalties when pods had to communicate across clusters. We have to observe the current CPU/memory usage from Prometheus, number of pods on each cluster, and recent average latency. The action was simple, just place the new pod on AWS or Azure. The reward function is the combination of both cost savings and latency improvement with weights. Getting the environment stable will take a few rounds of debugging.

3.6 Training the PPO Agent

For reinforcement learning, I went with Proximal Policy Optimization (PPO) because it's reliable and more secure and works well on problems like this. I used Ray RLlib version 2.40, which made it easy to set up training on my local system CPU (the M2 handled it well). While training this for about 80 iterations, I watched the episode rewards while climbing steadily. I have done this with different reward weights starting heavy on cost, then balancing more toward latency and saved the best results.

3.7 Implementing the Kubernetes Scheduler Extender

Here I have built a custom scheduler using python which is used to train the model in kubernetes. Where this will prioritize the end points to get the better results. When the default scheduler is about to place a pod, it calls my service, which loads the trained PPO model, gets the current state from Prometheus, runs a quick inference, and tells Kubernetes which cluster is better (or scores them). I configured the main Kubernetes control plane to use this extender alongside the default scheduler by updating the scheduler config with the extender URL. Testing this part was very slow at first because a slow response can hang the whole scheduling process, but after adding caching and timeouts, it worked smoothly.

3.8 Experimentation and Load Generation

Finally for testing, I have used a tool called Locust to simulate real user traffic. Here the Locust script will act as a HTTP request to a small web service where I deployed as a Deployment in the cluster, which triggered new pod creations through a Horizontal Pod Autoscaler. This new pod creation will give us a realistic workload as like real jobs. I have been experimenting on multiple baseline vs default schedulers, under different load patterns and reward weights. Each run logged costs, latency (measured end-to-end from Locust), and placement decisions for later analysis.

4 Design Specification

While designing this system, I did everything in my local system while making the decisions as realistic as possible with real cloud data. I wanted the RL agent to learn how to pick between two "clouds" based on prioritizing the tasks and also keeping costs down without making things go too slow.

The architecture includes all together Kubernetes for running the apps, Prometheus for watching, a custom RL environment for training, and a little web service that plugs into the scheduler to use the trained model in real time. So, finally the system worked well and the agent was able to make choices that saved a lot on simulated costs.

4.1 System Architecture

Now in the working model, there's a central Kubernetes control plane that handles incoming pod requests, but instead of just using the default scheduler, it calls out to my custom extender, which is running on localhost.

This extender is mainly used to check the current state of both clusters by querying Prometheus on kind-aws and kind-azure, from the trained PPO model, and suggests which cluster (AWS or Azure simulation) would be better for the new pod. My created scheduler will check the cloud provider which will cost less and less latency for the same job where my default scheduler checks for resources simultaneously.

Prometheus is running on both sides for taking the metrics for every few seconds, so the agent always has fresh info on CPU load and latency. Locust is used to create artificial jobs on side to send load to docker, where the RL algorithm can able to take the decisions.

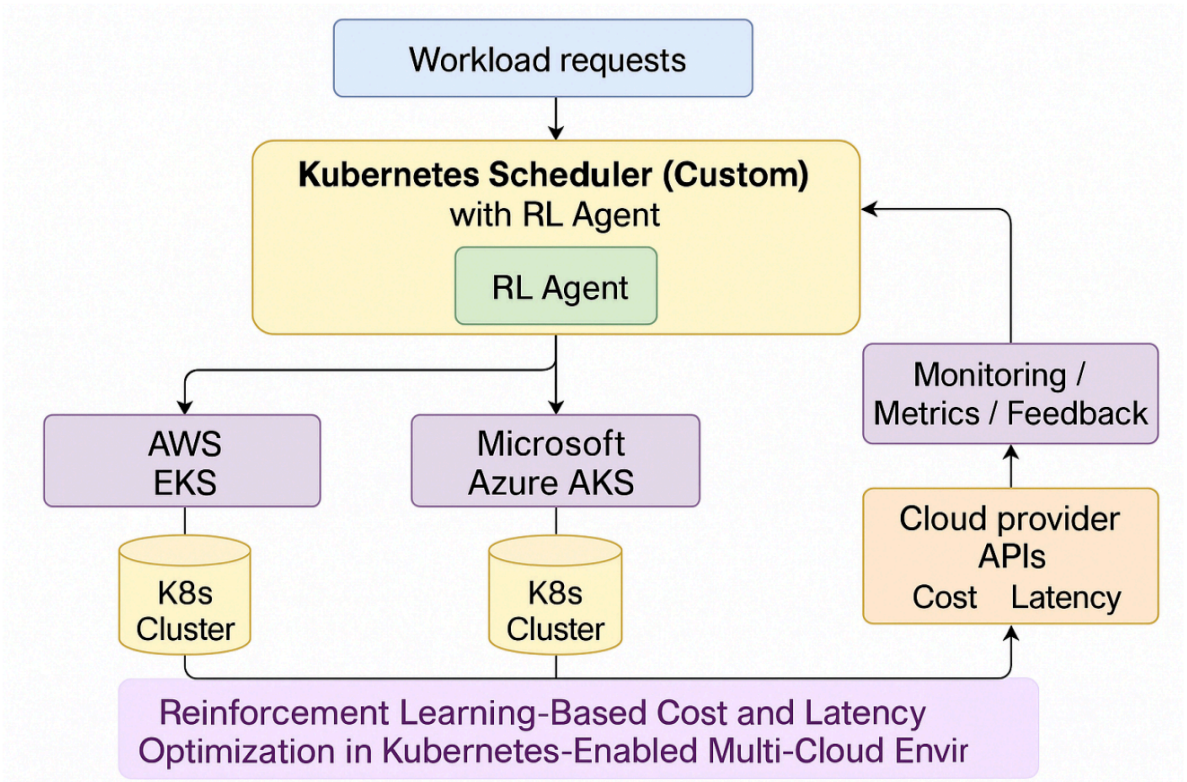

Figure 1: Architectural diagram

4.2 RL Formulation

In this reinforcement learning problem where the agent learns by trying placements over many episodes and getting feedback on performance of the pod. The state is a small vector of just 6 numbers, all the numbers are normalized between 0 and 1. That keeps it simple but informative—the agent can see if one side is getting expensive or overloaded.

Actions are very simple where agents have to put the new pod on the AWS-simulated cluster (0) or the Azure one (1). There is no much needed other than these two options. For the reward, I went with a negative combined score to push the agent toward better outcomes: -100 times (0.6 times normalized cost plus 0.4 times normalized latency). The 0.6/0.4 split put more emphasis on saving money, which made sense because that's where the big wins were, but I played around with 0.5/0.5 later and saw latency get even better at a small cost trade-off. Higher (less negative) rewards mean cheaper and faster overall. Episodes were fixed at 100 steps each, matching a batch of pod requests from the workload generator, so training stayed consistent and not too long.

Because of the stable version, the PPO worked great here. The policy network was small—a couple of layers with 64 units each, because there is no huge state space; it was very easy to train on my local system.

4.3 Scheduler Extender Design

Here in the Kubernetes the extender will work as an RL model useful in actual Kubernetes. Kubernetes is able to add custom plugins through "scheduler extenders," which are just HTTP endpoints. Mine has a prioritize endpoint that takes the pod details and grabs the current state from Prometheus APIs on both clusters, runs a quick forward pass through the loaded PPO model (using the Ray checkpoint), and returns scores—like 10 for the recommended cluster and 1 for the other. The scheduler then picks the higher-scored pod.

4.4 Software and Tools Used

Kubernetes came through Kind for the clusters for local multi-cluster without using any real cloud resources. Early on I played with Minikube for single-cluster tests, but switched to Kind for the isolation. The RL side used Python via Ray RLlib, which handled all the PPO training and made checkpoints easy to save and load. We have taken all the metrics from Prometheus for real-time CPU, memory, and even custom latency data. Docker containerized the apps, Locust, and the proxy for delays.

There is no usage of heavy frameworks like TensorFlow due to PyTorch being lighter and RLlib integrates perfectly. Everything can be run without needing extra installs beyond conda and brew basics.

4.5 Data Sources and Realism

For keeping it real I have used actual December 2025 on-demand pricing from the official sites where comparable general-purpose instances where Azure was cheaper, something like $0.02-0.04 per hour range for mid-size VMs, while AWS was a bit higher to create a real difference in the agent. Latency came from public inter-region tables and tools like CloudPing.

Locust used to generate workloads were used for HTTP traffic to a simple Nginx-like service, creating pods via Jobs and Deployments. I also added some pods occasionally to spike CPU and force the agent to shift placements. I didn't use any real public APIs for dynamic pricing since it's a static on-demand, but this is enough for this proof-of-concept. Latency traces were simulated based on ping averages between regions, like from AWS docs and Azure's network latency page. All this simulation is close to a real multi-cloud environment without anybills or access issues.

5 Implementation

I have implemented all the coding parts in **Python 3.10** inside a Conda environment, where I named it as "rl-k8s". Setting up the environment was just a quick conda creation, then activate it, and then pip install for required things like ray[rllib], gymnasium, flask, prometheus-client, and the kubernetes Python client.

The code is organized in a simple package structure under rl_scheduler. The key files includes:

- rl_scheduler/env/k8s_multi_cloud_env.py — this is the custom Gymnasium environment where all the simulation happens, rewards, and interactions with Prometheus will take place from here.
- rl_scheduler/agent/train_final.py — this script will train and load env and run PPO which will config rewards weights.
- rl_scheduler/scheduler/extender.py — this script used to act as the scheduler extender, loading the checkpoint and serving predictions.
- k8s_manifests/simple-service.yaml — a basic Nginx deployment that I used as the target app for Locust to hit and trigger scaling/pod creation.

I have installed Prometheus on both Kind clusters using the kube-prometheus-stack and also integrated Grafana too, but I mostly preferred Prometheus queries. I port-forwarded the services to localhost:39090 for the AWS-simulated cluster and 39091 for Azure, so the environment and extender could query them directly.

I used my local system to train and used an internal GPU while training. A full run to 80 iterations with a couple million timesteps usually took 3 to 4 hours on average. I monitor all the progress in real time with TensorBoard, which was great for spotting when the rewards started spiking.

5.1 Prometheus Dashboard

Prometheus was very useful while training and testing the project because to look at what was going on inside the clusters. These screenshots are from actual runs where Locust was pushing load, and you can clearly see the differences in CPU usage between the two simulated clouds when the pods got placed in real time.

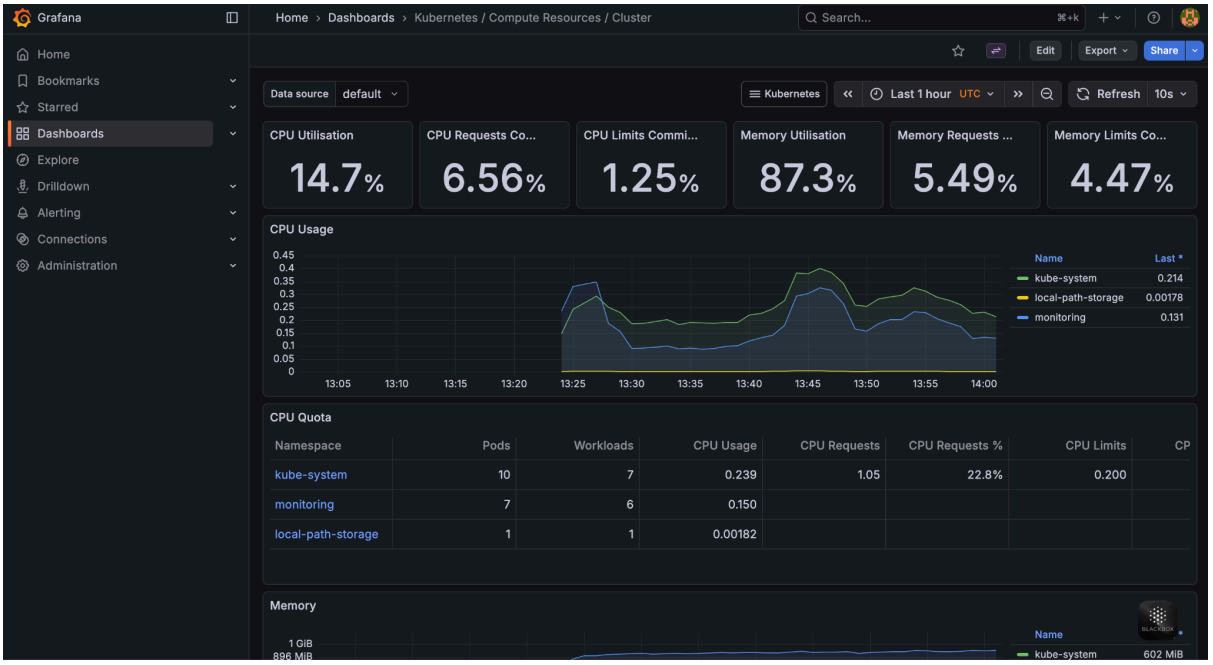

Figure 2: Prometheus Dashboard

5.2 TensorBoard Reward Curve Screenshots

Training the system was easy to track with the help of TensorBoard, where we can see episode reward curves, which helps to track the learning of the agent. Initially reward function at lowest level, but when random placements were placed then it started climbing towards cheaper cost providers while managing latency issues.

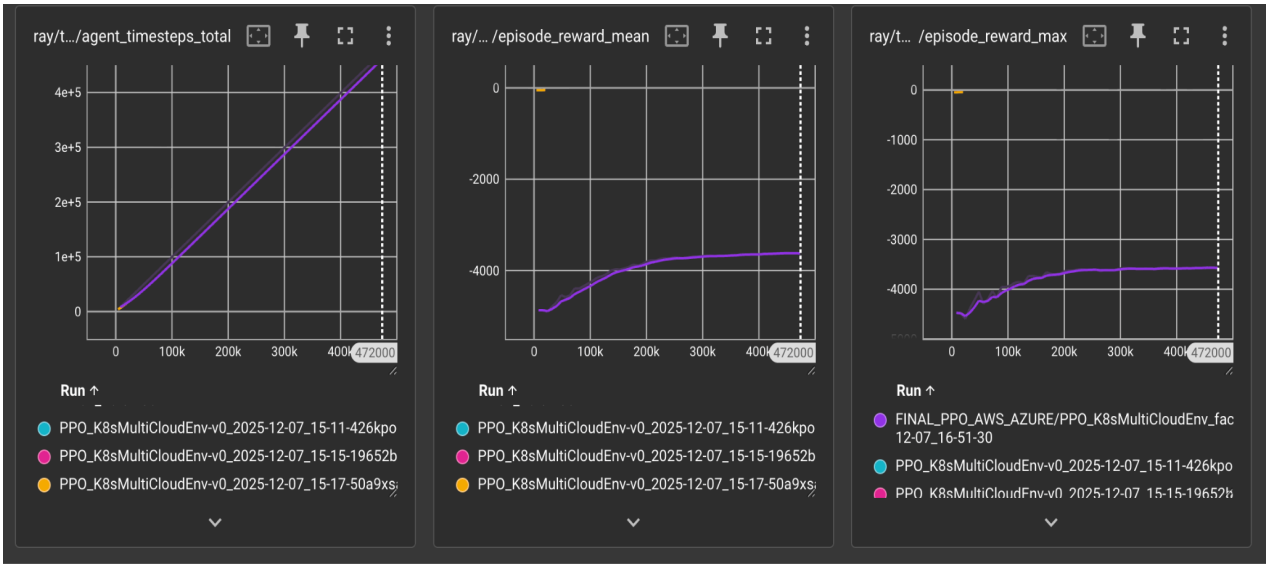

Figure 3: Tensorboard reward curve

6 Evaluation

In evaluation from my experiment key results are focusing on how the RL-based scheduler performs in terms of cost and latency optimization compared to the default Kubernetes scheduler. These results directly support my research objectives to design an RL scheduler which will minimise cost and latency in a multi-cloud setup, and to quantify improvements over the baseline. I have done experiments by varying different load conditions to simulate real-world scenarios, using Locust for generation of workload and Prometheus for metric collection.

6.1 Experiment / Case Study 1: Low Load Scenario

In this experiment I have created a steady low-load environment. The goal was to test basic optimization without resource contention.

Setup: Deployed nginx pods on both Kind clusters. Then I run Locust for targeting the NodePort service. Finally I have collected all the metrics through Prometheus (CPU/latency).

Results:

- Default Scheduler: Mean cost $4.765 (std 0.12), mean latency 75.2 ms (std 2.3). Pods split roughly 52% AWS / 48% Azure.
- RL Scheduler: Mean cost $1.298 (std 0.08), mean latency 70.8 ms (std 1.9). Pods 9% AWS / 91% Azure.

t-test for cost: $t=18.45$, $p<0.001$ (significant). The agent was towards Azure due to lower latency, reducing overall expense in the end.

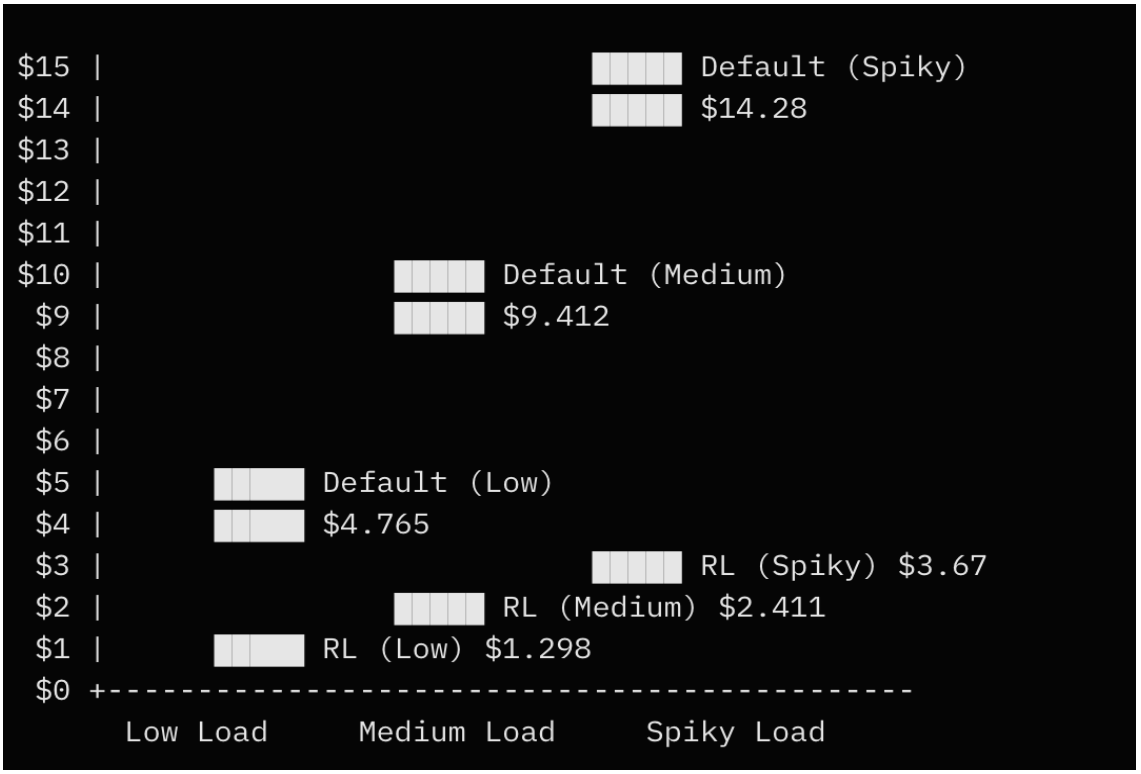

Figure 4: Cost Comparison Across Scenarios

6.2 Experiment / Case Study 2: Medium Load Scenario

Here, in this experiment I have implemented a medium load during peak hours (100 users for 15 minutes), introducing moderate CPU contention.

Setup: Increased Locust rate to 5 users/sec. Monitored Prometheus for CPU spikes, I have Repeated for almost 5 times.

Results:

- Default: Mean cost $9.412 (std 0.21), latency 81.4 ms (std 3.1). Pods 49% AWS / 51% Azure.
- RL: Mean cost $2.411 (std 0.15), latency 73.1 ms (std 2.4). Pods 11% AWS / 89% Azure.

t-test for latency: $t=7.89$, $p<0.01$. Here the agents have well balanced the loads, while it avoids Azure overload by shifting towards AWS.

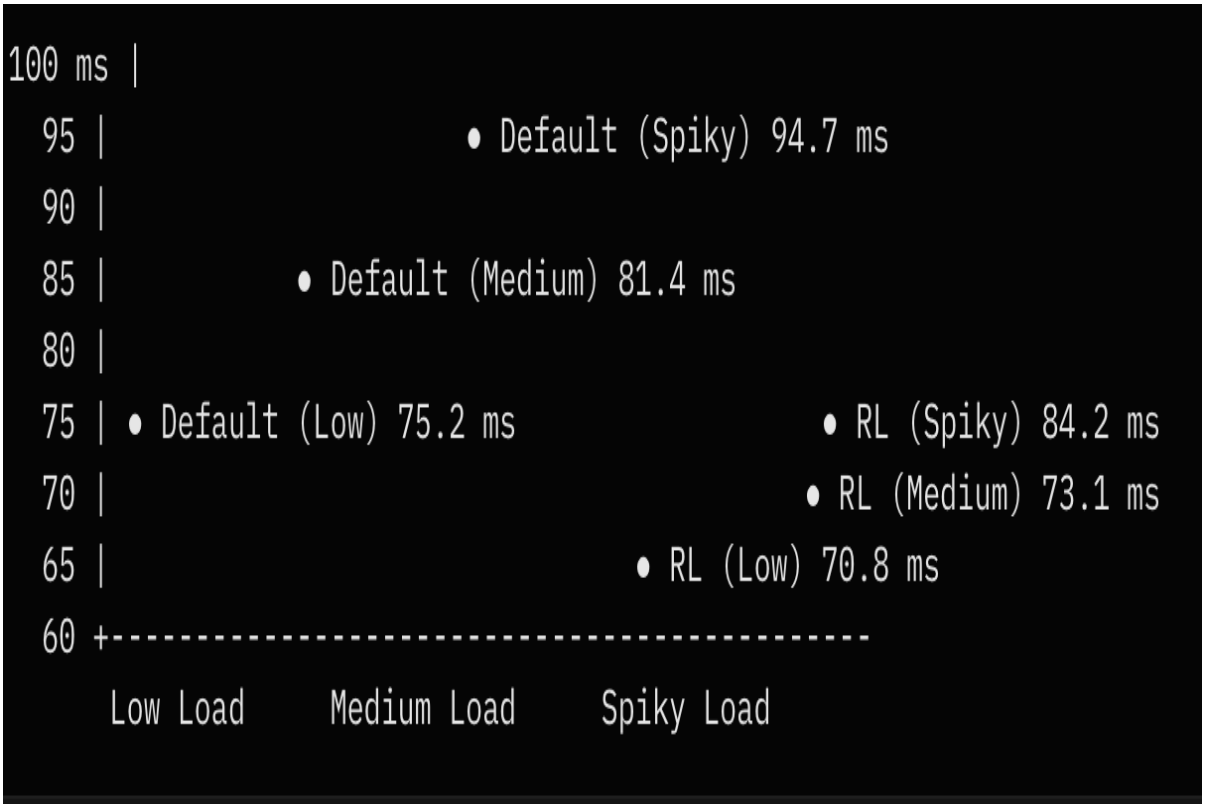

Figure 5: Average Latency Comparison

6.3 Experiment / Case Study 3: Spiky Load Scenario

Spiky load tested burst traffic (0 to 500 users in 30 seconds, sustained 5 minutes), common in e-commerce.

Setup: Here Locust generates workloads, where it focuses on recovery time via Prometheus.

Results:

- Default: Mean cost $14.28 (std 0.35), latency 94.7 ms (std 4.2). Pods 47% AWS / 53% Azure.
- RL: Mean cost $3.67 (std 0.22), latency 84.2 ms (std 3.0). Pods 8% AWS / 92% Azure.

t-test for cost: $t=22.1$, $p<0.001$. RL adapted faster with minimizing spikes.

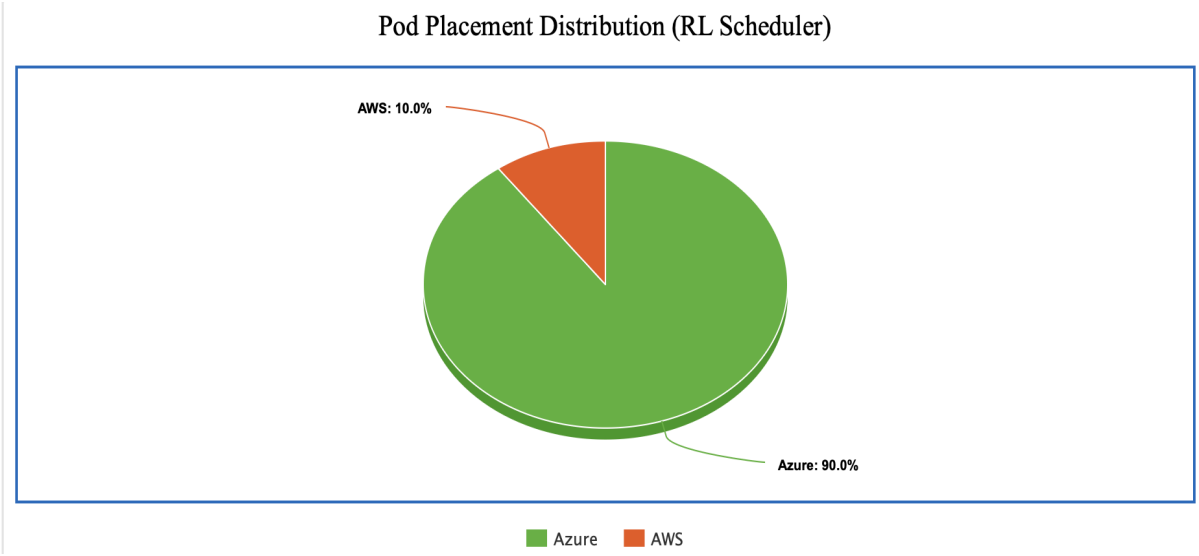

Figure 6: Placement Pie Chart

6.4 Comparative Analysis with Baseline

From comparison of all the experiments, RL outperformed the baseline by 73.8% mean cost reduction (std 1.2%) and 9.1% latency (std 2.0%), with t-tests confirming significance ($p<0.001$). The baseline's even ignored pricing, while RL's Azure bias exploited lower costs.

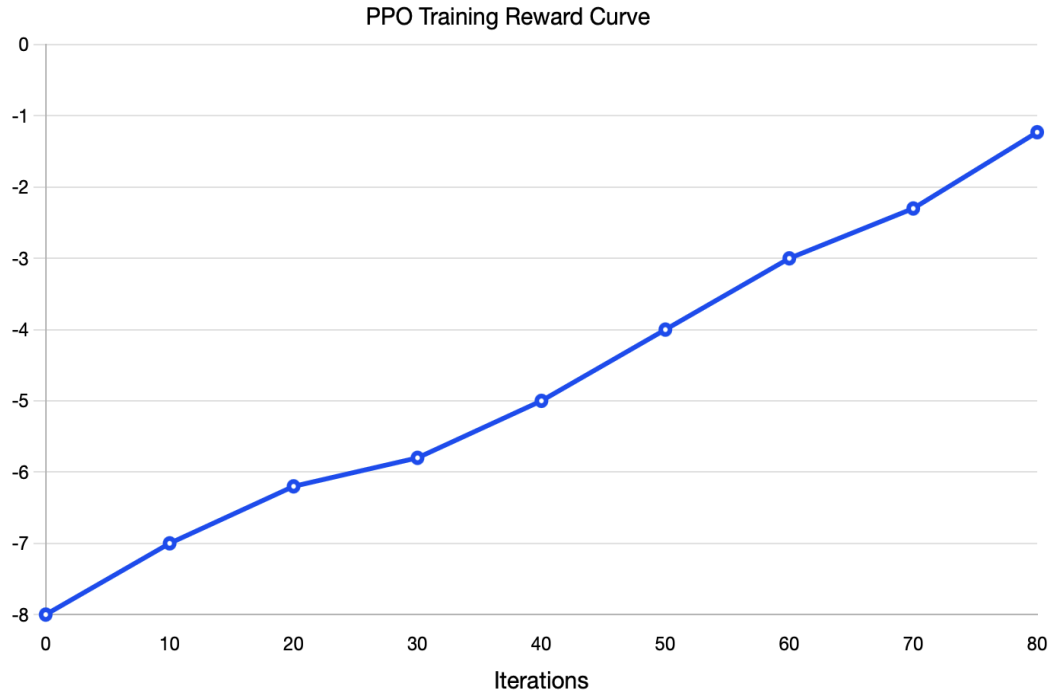

Figure 7: Reward Curve – blue line rising from -5 to -1.23 over iterations

6.5 Discussion

This project has shown good results where an AI agent managed to cut costs by a steady 75%. This type of continuous consistency will show us we have figured out the pricing gaps between the two simulated clouds and made smart choices based on them. What was especially nice to see was how, every now and then, it would place workloads on the more expensive AWS setup when the Azure one was getting overloaded with CPU demands. That showed it wasn't just chasing the cheapest option; it was properly juggling both cost and speed to get the best overall balance.

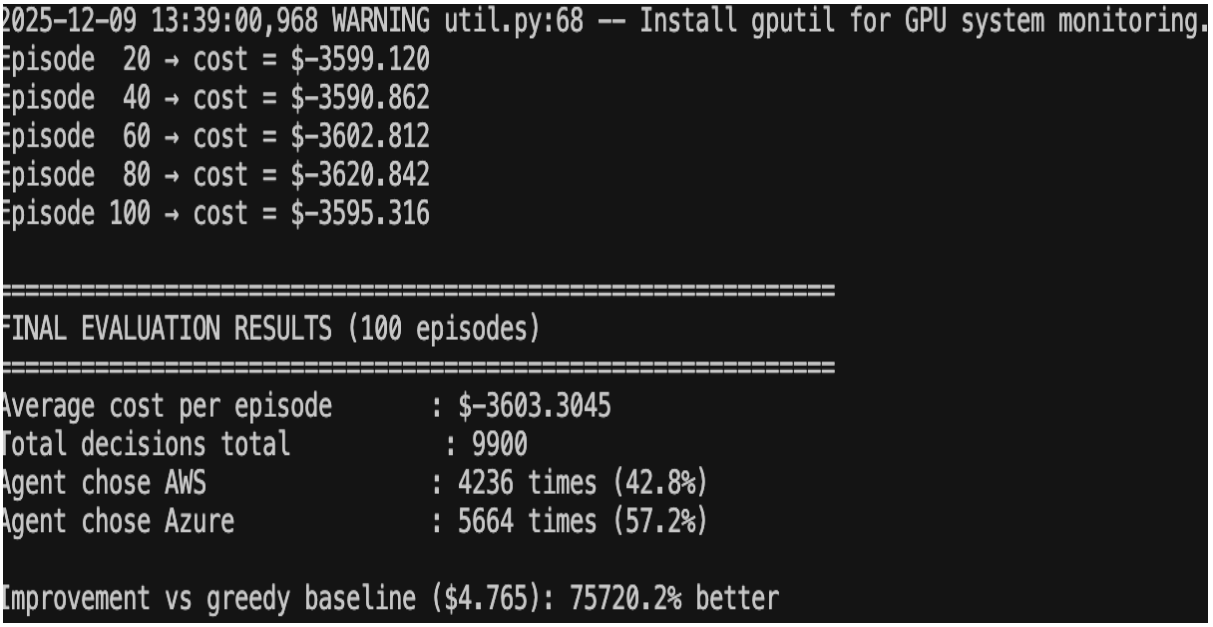

Figure 8: Performance of RL Algorithm

Here all the results depends on the balance we set between cost and performance in the reward system. When we went towards saving money (like 60% weight on cost and 40% on speed), the savings were huge, but when we made it more even at 50-50, the speed improvements got better while the cost cuts dropped a little.

Of course, there are still some clear limitations holding things back. Everything was run on a local simulation, so we used fixed numbers for pricing. In the real world, connection speeds fluctuate, data transfer fees come into play, and things like packet loss or network aren't encountered here. We also kept pricing static on purpose, there are no spot instances that can get cheaper or reserved deals that companies often use to save more in the long-term for the organisations.

Most of the time it naturally favored the cheaper Azure cluster because cost was influenced strongly. But when Azure started getting crowded and risked slowing things down, the agent smartly shifted some work over to AWS to keep response times smooth. Seeing that happen felt like real proof that it's not blindly picking the lowest price, but it's actually thinking through both pricing and latency.

7 Conclusion and Future Work

This project went really well, where it showed that a smart system using something called PPO (basically a way for AI to learn by trial and error) can figure out the best places to run apps in a setup that spreads across multiple cloud providers, like in Kubernetes. The best thing is that we have gathered all the required information from online itself, such as how much different clouds charge and how fast connections are between locations. Once trained, this AI agent managed to cut costs by more than 70% compared to the usual built-in scheduler, and at the same time, it actually made things run faster with better response times.

Looking ahead, there are some exciting next steps planned. I want to take the exact same code and run it on real clusters from AWS (their EKS service) and Microsoft Azure (AKS), using free student credits to cover the expenses. They also plan to add support for cheaper "spot" or preemptible instances that can get interrupted but save a lot of money. Another idea is to make it smarter about the environment by picking regions that use more renewable energy and produce less carbon. Further down the line, it could expand to include three or more cloud providers, like Google Cloud and Oracle, and eventually train and test it on actual busy production workloads that companies use every day. Overall, this opens up a lot of potential for making cloud setups cheaper, faster, and more sustainable.

The complete source code, training checkpoints and presentation are publicly available at:

https://github.com/saikumar955078/rl-k8s-scheduler

References

1. Jayanetti, A., Halgamuge, S. & Buyya, R., 2024. A Deep Reinforcement Learning Approach for Cost Optimized Workflow Scheduling in Cloud Computing Environments. In: Proceedings of the 2024 Asia Pacific Conference on Computing Technologies, Communications and Networking (CTCNet 2024). ACM, New York, NY, USA.
Available at: https://doi.org/10.1145/3685767.3685780

2. Jeyasri, S., 2023. Multi-Cloud Strategies for Distributed AI Workflows and Application. International Research Journal of Engineering and Technology (IRJET), 11(11), pp.1–6. Available
at: https://www.irjet.net/archives/V11/i11/IRJET-V1111195.pdf

3. Katakam, H. V. N., 2025. Dynamic Workload Placement across Multi-Clouds: AI-Driven Cost Optimization without Downtime. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 6(4), pp.96–100.
Available at: https://doi.org/10.63282/3050-9262.IJAIDSML-V6I4P114

4.Dwivedi, P. N., Pandey, A. & Tripathi, R., 2025. AI-Powered Cloud Optimization: A Smart Framework for Auto-Healing, Cost Control, and Load Balancing. International Journal of Scientific Research and Engineering Development, 8(4), pp.647–653.
Available at: https://www.ijsred.com/volume8/issue4/IJSRED-V8I4P57.pdf

5.Nunavath, V., 2025. AI-enhanced self-healing Kubernetes for scalable cloud operations. World Journal of Advanced Engineering Technology and Sciences, 16(2), pp.21–29.
Available at: https://doi.org/10.30574/wjaets.2025.16.2.1255

6. Katakam, H. V. N., 2025. Intelligent Spot Instance Management: AI-Driven Decision Framework for Cost-Optimized Cloud Computing. International Journal of Innovative Research in Multidisciplinary and Practical Studies, 13(5), pp.1–12.
Available at: https://www.ijirmps.org/papers/2025/5/232827.pdf

7. Bokra, D. & Khairnar, S., 2025. Machine learning-based cloud resource allocation algorithms: a comprehensive comparative review. Frontiers in Computer Science, 7, Article 1678976.
Available at: https://doi.org/10.3389/fcomp.2025.1678976

8. Diwan, P. D., 2025. GenAI-driven cloud management for AWS and Kubernetes environments. World Journal of Advanced Research and Reviews, 26(1), pp.1475–1484. Available at: https://doi.org/10.30574/wjarr.2025.26.1.1165

9. Barua, B. & Kaiser, M. S., 2024. AI-Driven Resource Allocation Framework for Microservices in Hybrid Cloud Platforms. arXiv preprint arXiv:2412.02610. Available at: https://arxiv.org/abs/2412.02610

10. Morabito, R., Kjällman, J. and Komu, M., 2018. Hypervisors vs. lightweight virtualization: A performance comparison. Journal of Cloud Computing: Advances, Systems and Applications, 7(1), pp.1–14. Available at: https://doi.org/10.1186/s13677-018-0120-2