A Proactive Hardware-Aware Pod Migration Approach for Real-Time Applications in Edge Clouds

MSc Research Project

Cloud Computing (MSCCLOUD1_A)

Majid Shahbaz

Student ID: x23332085

School of Computing

National College of Ireland

Supervisor: Yasantha Samarawickrama

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Majid Shahbaz		
Student ID:	x23332085		
Programme:	M Sc. Cloud Computing	**Year:**	2024
Module:	Research Project		
Supervisor:	Yasantha Samarawickrama		
Submission Due Date:	11/08/2025		
Project Title:	A Proactive Hardware-Aware Pod Migration Approach for Real-Time Applications in Edge Clouds		
Word Count:	10500	**Page Count**	28

Signature: Majid Shahbaz

Date: 10/08/2025

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.	□

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

A Proactive Hardware-Aware Pod Migration Approach for Real-Time Applications in Edge Clouds

Majid Shahbaz

x23332085

Abstract

Modern container orchestration platforms like Kubernetes have radically changed the way applications are deployed in hybrid and edge environments. While Kubernetes is very scalable and manages containers automatically with great efficiency, it does not natively support live pod migration which is an important feature for real time and latency-sensitive applications. In standard deployments, pods are terminated and restarted on other nodes during hardware failures or maintenance, causing unwanted service interruptions in systems that require continuous availability and low latency. In typical deployments, pods are terminated and restarted on different nodes during hardware failures or maintenance, causing unwanted service interruptions in systems that require continuous availability and low latency. To address these limitations and challenges, this research introduces a Proactive Hardware-Aware Pod Migration approach that designed to overcome these limitations by using deep reinforcement learning (DRL) to intelligently predict hardware resource bottlenecks and initiate live migration with the help of Checkpoint/Restore in User space (CRIU) ahead of failures. This approach considers hardware heterogeneity across edge nodes such as differences in CPU architecture and memory capacity – to determine optimal migration paths that maintain SLA compliance.

The experimental verification using actual deployment data from Amazon EKS environments shows considerable performance gains: 67.8% decrease in response time, 63.8% decrease in SLA violations, and 90% decrease in downtime. The system provides 100% migration success rate while ensuring proactive decision making in 85% of the migration cases in multi-tenant edge computing environments. Statistically significance tests validate all enhancements are statistically significant ($p< 0.001$) with large effect sizes, showing the DRL-based solution achieves an effective intelligent pod migration in edge clouds.

Keywords — Kubernetes, Pod Migration, CRIU, Edge Computing, Deep Reinforcement Learning, Hardware-Aware Scheduling, Real-Time Applications.

1. Introduction

Kubernetes is widely acknowledged as the best containerized applications across cloud native infrastructures. It enables container scheduling, scaling, and orchestration at scale. However, Kubernetes has no native support for live pod migration, especially in multi-tenant edge environments. In most cases, pods are terminated and then re-created on different node during failures or scheduled maintenance, causing unacceptable disruptions. Existing research in pod migration mainly has a reactive approach migration is only triggered whenever a failure or resource bottleneck has occurred. Furthermore, many solutions overlook the hardware heterogeneity of edge nodes, and it has not included the unique demands of the real time workloads or fairness in multi-tenant environments. These gaps lead to inefficient resource utilization, unfair allocation, and higher response times.

Real-time applications, like augmented reality, industrial robotics, autonomous driving, and the remote medical services require extremely low latency and continuous service availability. Traditional cloud-based architectures often fail to meet these requirements due to high communication latency and centralized of computing resources. As client devices are physically distant from cloud data canters, even small delays in communication can degrade the performance and reliability of time sensitive services.

Edge computing focuses on addressing these challenges by moving data processing closer to the source. However, edge environments are typically resource constrained and heterogeneous in nature. Edge nodes vary in CPU, memory, and I/O capabilities. This variability introduces complexity when managing and migrating containerized workloads, particularly for real time services where uninterrupted performance is critical.

Although other studies have tried to overcome these difficulties and complexities with proactive and smart scheduling schemas. Ali et al. (2024) designed ProKube, a telemetry-based scheduling orchestration framework that used predictive metrics to schedule pod migrations prior to occurrence of bottlenecks. But they didn't fully focus on all these factors like Hardware aware, multi-tenant and DRL based pod migrations.

Likewise, Li Ju et al. (2021) developed the proactive Pod Auto-scalar (PPA) that utilizes forecasting models like LSTM to forecast future workloads and scale Kubernetes applications in advance. These tools showed better response time and less wastage of resources in edge computing environments. Although their method does not directly address live pod migration or hardware-level heterogeneity. To facilitate uninterrupted service continuity, Schrettenbrunner et al. (2020) investigated live checkpointing and restoration for pod migration in Kubernetes through CRIU (Checkpoint and Restore in User space). This research presented the notion of "Migrating Pods" as bespoke Kubernetes resources which allowed the stateful pods to be cloned and then migrated across nodes with minimal disruption. Although this method substantially minimizes downtime, it is still susceptible to ineffective migration choices when utilized in standalone capacity, specifically in complicated, multi-node edge deployments.

However, Deep Reinforcement Learning (DRL) has been promising in solving dynamic scheduling and migration issues in mobile and edge networks. Cui et al. (2024) proposed an AW-DDPG based framework for facilitating intelligent task migration in mobile edge scenarios based on real-time resource availability, network load and the application demand. Their solution chooses the best edge nodes to execute tasks, highlighting the flexibility and context awareness of DRL models in decentralised computing systems.

To communicate these gaps in current pod migration frameworks, this study seeks to answer the following key question:

Research Question – *How can proactive, hardware-aware pod migration using deep reinforcement learning improve latency, fairness, and resource utilization for real-time applications in multi-tenant edge cloud environments?*

To answer this question, this research proposes a **Proactive Hardware-Aware Pod Migration** Approach for Kubernetes in multi-tenant edge clouds. The system will use Deep Reinforcement Learning (DRL) such as Proximal Policy Optimization (PPO) or Deep Deterministic Policy Gradient (DDPG) to predict workload patterns and proactively migrate pods before failures occur. The decision-making process will be hardware-aware, ensuring that each migration aligns with the resource requirements of the pod and the capabilities of the target node. The solution aims to reduce downtime, enhance SLA compliance, and improve edge resource utilization by computing intelligent learning-based decisions with platform-level orchestration tools. The study proposes to design, implement, and evaluate this system through simulation-based benchmarking under varying load and configurations.

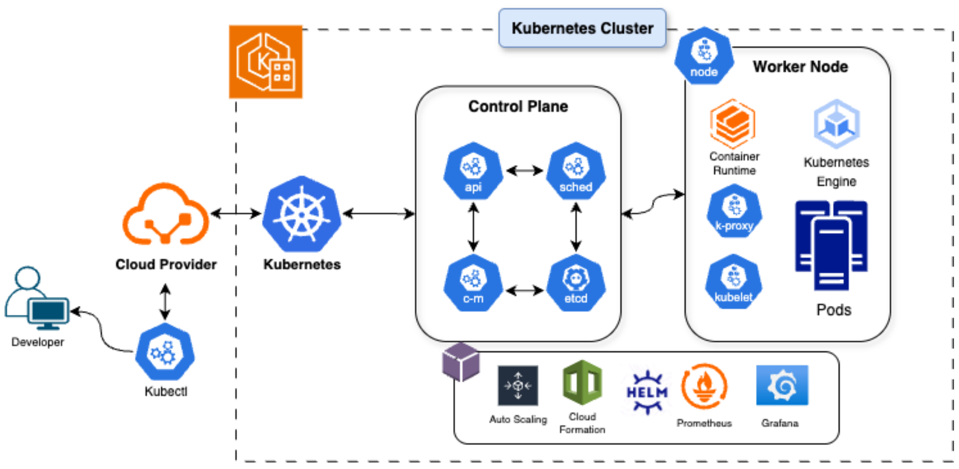


Figure1. Kubernetes and Cloud Architecture

Figure 1 illustrates a design level diagram of this proposed solution, describing the interaction between elements of Kubernetes and cloud. The control plane of Kubernetes manages the cluster elements such as API server, scheduler and the worker nodes contain the pods, K-proxy, container runtime, Kube-let and Kubernetes engine. When a DRL agent integrates into the architecture, it functions in a proactive manner to determine preeminent migration actions autonomously based to determine optimal acting as the core cloud services manager.

The main contributions of this research as following:

- **A DRL based pod migration framework** that enables hardware awareness, SLA compliance and telemetry from across the system to learn optimal migration policies in edge environments.
- **Implementation of a live migration pipeline** through Checkpoint/Restore in User space (CRIU) to achieve stateful pod migration with minimal disruption of service.
- **Modular design system** integrates monitoring (Prometheus, Grafana), decision making agent, and pod migration executor.
- **Extensively handling of resource** allocation during pod migrations such as CPU capacity availability, Memory availability and nodes compatibility.
- **Cooldown logic and built in safeguards** lower the risk of duplicate migrations and improves the fault tolerance in multi-tenant clusters.

This report is organized into different sections. Section 2 gives a literature review of existing pod migration and intelligent scheduling methods. Section 3 gives the research methodology and experimental design, followed by Section 4 that presents the system architecture and design implementation process comprising the PPO model development and deployment. Section 6 gives the evaluation results and performance analysis, while Section 7 concludes the major contributions and future work directions.

2. Review of Literature

This current review section will assess previous studies based on their techniques, optimal resource allocation and utilization and evaluation metrics for the pod migrations. Pod migration is one of the main factors affecting service reliability while performing in multi-tenant heterogeneous edge networks hosting real time applications. This section presents a detailed review of the most relevant contributions in the field by comparing their work based on hardware awareness, multi-tenancy, fault tolerance, Quality of service (QoS) management and the usage of Deep Reinforcement Learning in their work.

2.1. State-Of-the-Art in Kubernetes Pod Migration

Study by **Indrani et al.** in 2024 proposed a pod migration mechanism using Persistent Volumes (PVs) to conserve the current state of containers during migration and loading them on retrieval during deployment with the purpose of minimizing the downtime of pods. Compared to other migration methods including Kubernetes default, which reimages all containers, their proposed solution redeploys selected active containers, saving cold start and resource usage. This design includes a migration controller that triggers state capture and pod redeployments with only live containers. Result from experiments show significant CPU and Memory usage optimization as compared to standard pod restarts. The paper by **Poggiani el al.** in 2025 introduced a Kubernetes native solution to perform live migration of single and multi-container pods across geographically remote edge clusters. Their study focuses on serverless platforms and used CRIU for saving state of container with Liqo, a multi cluster orchestration framework, to establish peer links and migrate runtime state of containers. Their solution is fully aligned with Kubernetes API and doesn't use intermediary nodes through direct transfer of check points between clusters. Proposed solution tested in multi cluster testbed by created a customized source and destination migration operator to deal with checkpointing, transfer and restoration. The research proved the feasibility of live migration in serverless edge systems but also represents the performance bottlenecks in restoration in multiple containers.

Schrettenbrunner, F. et al. in 2020 examined the integration of live migration techniques into Kubernetes with checkpoint/restore (CRIU) mechanism. They found a strong vulnerability in how Kubernetes has always provided pod rescheduling method by killing and recreating pod elsewhere, which introduces service downtime to applications no essential high availability. They proposed a cloning based migration technique where pods are cloned and cut over in a way that imitate live migration. To enable stateful migration operation in Kubernetes, a custom resource called Migrating Pod was introduced. A controller from the Kubernetes API implemented this process in an automated fashion. The proposed solution indicated the viability of migrating pods in containerized systems. Although these studies have made significant contributions to address pod migration challenges, but many gaps remain unattended. The studies reviewed above has made significant development, yet they have following limitations:

2.1.1. Study by Indrani et al. in 2024 has described a method to reduce downtime by means of persistent volumes and partial pod scaling. However, their work does not include predictive scaling or proactive migration with hardware awareness, which are important to deal with real time load variations in edge cloud scenarios.

2.1.2. Poggiani et al. in 2024 introduced a system that supports multi container migrations in serverless Kubernetes environments, their work emphasizes performance bottlenecks during container restoration. Their study does not consider migration prioritization or predictive workload management for timing and resource optimization with the help of Deep Reinforcement Learning.

2.1.3. Schrettenbrunner et al. in 2020 experimented with live migration with checkpoint/restore (CRIU) but focused primarily on pod duplication and stateful state transitions. Their study does not address proactive pod migrations approach with hardware awareness.

2.2. Intelligent Scheduling and Pod Migration Strategies

In 2024 a study by **Ali, B. et al.** introduced a new system known as ProKube, an orchestration tool that employs prediction to orchestrate pod migrations in cloud environments. Prokube also allows prediction-based relocation of containers to avoid latency and resource bottlenecks, leverage telemetry-based scheduling. This study covered several important gaps in Kubernetes pod migrations and resource management. However, there are still areas that further require improvement and optimization such as Deep Reinforcement based pod migrations along with hardware awareness in multi-tenant edge cloud environments. **Li Ju et al.** in 2021 proposed an Proactive Pod Auto-scaler (PPA) approach for Kubernetes managed edge computing applications. PPA forecasts workloads using multiple users provided metrics, which enables proactive scaling of applications. For those edge applications which require more CPU, PPA introduced improved resource usage and application performance compared to Kubernetes default auto scaler. Proactive Pod Auto-scaler (PPA) is evaluated using synthetic (Random Access) and real (NASA) workloads with minimal response time and reduced wastage of resources. Experiments confirm that LSTM based models performs notably better in edge deployments as compared to traditional approaches.

The research **Cui et al.** (2024) introduced a DRL based task migration framework for mobile edge networks that employs an adaptive Weighted Deep Deterministic Policy Gradient (AW-DDPG) algorithm. This work facilitates the selection of the most suitable network between multiple users in accordance with the load, user demands and resources available. The system automatically learns about the optimal edge nodes to execute task migration to save cost and latency. However, this proposed system does not operate on Kubernetes environments, and it does not support pod level state management, but it presents an intelligent way to manage task and resources using Deep Deterministic Policy Gradient (AW-DDPG).

Kumar et al. (2024) proposed a proactive autoscaling framework in Kubernetes which is based on predictive AI models to dynamically allocate and manage resources in containerized environments. This system also merges Autoregressive Integrated Moving Average (ARIMA) model, Long short-term Memory (LSTM) model, Bi-LSTM and transformer models into Kubernetes using custom resource and operator balancing process. This system efficiently handles the variable workload asynchronously using coroutines and resolves the issues of under and over provisioning to a large extent. Transformer model produced minimum prediction error with good accuracy and usage efficiency over NASA-HTTP dataset. Pashaeehir at el. (2025) presented KubeDSM which is Kubernetes based framework that improves dynamic scheduling and live migrations in cloud clusters. It addresses resource breaking with batch scheduling and live migration techniques in intra-edge, edge-to-cloud, and cloud-to-edge scenarios. The framework has addressed maximizing resource utilization while providing the Quality of Service (QoS) for latency sensitive applications. KubeDSM distributes the load efficiently among distributed environments to support workload migrations. The system tested across different edge deployments and obtained higher utilization along with consistent scheduling performance through standard methods. This makes the system usable in scalable and dynamic infrastructures.

However existing studies have made valuable contributions to address pod migration challenges in an intelligent way. But they still did not cover all the areas to improve the better migrations by the help of Deep Reinforcement Learning Models. Current studies have following limitations:

2.2.1. Proposed work by Ali et al. (2024) presents an effective proactive migration and scaling framework called ProKube. However, it does not support stateful pod migrations with Deep Reinforcement Learning or enforce real time SLA guarantees, which is extremely important for latency sensitive applications. Further their system lacks the hardware awareness like CPU and Memory.

2.2.2. The proposed study Li Ju et al. (2021) proposed an active pod auto-scaler based on predictive models like LSTM to scale but lacks pod migration or dynamic placement shifting between clusters. This study focused to scale pods not to address latency or other components like hardware awareness while migrating pods.

2.2.3. The study by Cui et al. (2024) introduced a performance aware task migration method based on DRL for edge mobile applications, but the system does not use Kubernetes and skips container scheduling or management of pod level state.

2.2.4. Kumar et al. (2024) proposed scaling solution with the help of predictive AI models, but their work still needs to auto scale and lacks support for pod relocation or migration among nodes or clusters. System's flexibility limits because of not integrating with container state management under edge load changes.

2.2.5. The research done by Pashaeehir el al. (2025), KubeDSM, improves batch scheduling and migration among edge clusters but it lacks the state aware scheduling or live resource profiling.

Several studies focused on pod migration, autoscaling and offloading in cloud and edge environments, but none of them provide a full solution that cover stateful pod migrations, hardware awareness, proactiveness and SLA guarantee in a Kubernetes native environment. Current studies either lack support for managing pod state such as (ProKube), are not executed within Kubernetes (e.g., DEL -based task migration techniques), or depends on scheduling which is called rule-based scheduling with leaning based adaption. Further, most of the studies did not consider hardware diversity, real time constraints and multi-tenant fairness, which are essential for modern AI based applications run at edge environments. My work fills that gap by suggesting a Deep Reinforcement Learning (DRL) driven pod migration solution that takes intelligent, hardware aware decisions before any failure occur. My approach analyses node level resource requirements of each pod before migration. Integration of Deep Reinforcement Learning models (DRL) such as Proximal Policy Optimization (PPO) into Kubernetes cluster, the system observes real time pods and nodes resources and learns in real time for the optimal migration policies under varying conditions.

3. Research Methodology

This research uses a quantitative experimental approach to design, develop and evaluate a proactive hardware-aware pod migration approach for real time applications in multi-tenant edge cloud environments. The research methodology consists of actual data retrieval from Kubernetes clusters combined with up-to-date machine learning techniques, specifically Deep Reinforcement Learning (DRL), in developing an intelligent migration decision system that learns optimal policies based on the continuous experience with the cluster environment.

The research methodology is organized into sequential steps with a systematic approach that begins with comprehensive data collection from production like Kubernetes clusters, going through advanced data preprocessing and feature engineering, and ending with training and deployment of a PPO-based DRL agent. Experimental design guarantees statistical validity through controlled test environments with Mini-Kube for development and Amazon EKS for production-scale verification with multi-tenant contexts to reflect real-world multi-tenancy circumstances. The methodology in specific involved comprehensive performance evaluation against the established baseline migration strategies including reactive responsive threshold-based migration strategies and default scheduling based on Kubernetes. The process prioritizes reproducibility with automated data collection pipelines, version-controlled model artifacts.

3.1. Problem Identification

The first important step in this study was to create a clear description for the issue of pod migrations in Kubernetes environment. A detailed analysis of the related work helped to identify the issues and deficiencies that Kubernetes based systems have related pods scaling and migrations by focusing on the nodes resources. The objectives and scope of this research is to design a system that should be a Proactive, Hardware Aware, Deep Reinforcement Learning based system capable of performing in multi-tenant spaces.

3.2. Experimental Setup on EKS Cluster

This research methodology included several crucial procedures. Initially whole setup was designed and configured locally by using Mini-Kube, which is a local Kubernetes. Further all the required plugins were installed on Mini-Kube cluster locally such as Prometheus with the help of Helm. To test and deploy the testbed environment onto a production-like infrastructure, Amazon Elastic Kubernetes Service (EKS) cluster was created using eksctl. The cluster has three worker nodes with resource heterogeneity to mimic hardware heterogeneity.

Following steps including in the setup:

- **Created** three tenant's namespaces (tenant-a, tenant-b, tenant-c)
- **Deployed workloads**:
    - ffmpeg-workload (video encoder, high SLA)
    - mqtt-broker (sensor data stream, medium SLA)
    - ngnix-pi (web server, low SLA)
    - Stress pod deployment for testing purpose
- **Monitoring Stack:**
    - Prometheus, Node Exporter and Grafana were installed via Helm.
    - All the Prometheus Metrics endpoints were exposed through port forwarding for development and debugging.
- **RBAC and Service Account:**
    - A separate Service Account was setup for migration-agent with the required Cluster Roles and binding to enable eviction, migration and metrics access for all namespaces in the cluster.

3.3. Data Collection Strategy

The information gathering framework serves as the core of the intelligent migration system, utilizing an advanced multi-agent system to efficiently collect comprehensive cluster state information from different modules in the Kubernetes ecosystem. The proposed approach for monitoring stratum provides observability for hardware, containers, and applications while preserving the consistency and temporal alignment critical for effective DRL training. Prometheus collected fine-grained metrics, including:

- **Pod-level:** timestamp, namespace, pod, node, workload_name, cpu_millicores, memory_bytes, sla, type
- **Node-level:** timestamp, node, cpu_usage_percent, memory_usage_percent, load_1min
- **Merged:** timestamp, namespace, pod,node, workload_name, cpu_millicores, memory_bytes, sla,type, cpu_usage_percent_node, memory_usage_percent_node, load_1min

3.4. Model Selection

3.4.1. Proximal Policy Optimization (PPO) model

Pod Migration problem in multi-tenant edge environments with hardware awareness introduces unique challenges that orient perfectly with the capabilities of Proximal Policy Optimization (PPO), which is a reinforcement learning algorithm. PPO model was explicitly chosen for this research due to many reasons that make it perfectly suited for the dynamic, real-time nature of systems.

3.4.2. Alternative Methods

Deep Deterministic Policy Gradient (DDPG): While DDPG is advantageous for continuous action space capabilities, versatile, and successful for robotics applications, it is more difficult to justify using DDPG in Kubernetes applications because of its off-policy nature, significant sensitivity to hyperparameter tuning along with the restrictions of the production environment or use case. Kubernetes applications or use cases, especially edge computing environments require stability in the operationalization of migration decision making as part of a migration policy.

Deep Q-Network (DQN): While DQN has strong theoretical justifications and represents a solid baseline for discrete action space applications, it lacks the detailed policy representation required for multi-objective optimizations associated with pod migration scenarios. Additionally, the value-based approach creates relatively static decisions across the available action space, limiting the opportunity for detailed policy representation in the considerations or contexts associated with hardware heterogeneity and performance requirements imposed by SLAs. Advantage Actor-Critic (A2C/A3C): These methods do have solid baseline performance characteristics and computational efficiencies, however, neither method uses a more advanced policy update approach that is sensitive to the possibility of significant policy changes that can impose instability on cluster operations within the production environment.

Trust Region Policy Optimization (TRPO): While TRPO supports theoretical guarantees for policy improvement, TRPO is computationally costly and converges at a slower rate making it more sophisticated or difficult to utilize in an application. This hardship is evident when one considers the decisions which can emerge during migration processes, especially in dynamic edge computing environments.

3.4.3. Why PPO is excellent for Pod Migration

Proximal Policy Optimization (PPO) handles the critical problem that arises from learning stable policies in environments, where potentially poor decisions can influence the performance of the entire system. Unlike other policy gradient methods that may suffer from large, destabilizing updates to the policy, PPO learns by operating using a clipped surrogate objective that prevents the policy from shifting too much from one update to the next. This is a necessary feature in production environments of Kubernetes where aggressive migration policies can disturb service availability and violate SLA requirements.

PPO is an on-policy learning algorithm, the agent learns from the most recent experiences, which helps model to continuously adjust the cluster's states change in relation to an SLAs condition. In an edge computing way, the operational resource availability and workload patterns vary continuously, this adaptability makes the system to maintain optimal performance regardless of dynamic operational conditions of workloads. Further, PPO's sample efficiency achieves effective learning based on the limited number of migrations events, incorporated because the increase or excessive usage of migrations can be detrimental to the overall systems stability. Actor-critic architecture supports two main advantages. First, the actor network learns how many migration policies are executed with multiple competing objectives, while critic network provides stable value estimates that reduce variance related to the policy updates.

3.4.4. Selection Criteria for Algorithm

PPO algorithm was selected based on weighted criteria listed below:

- Stability in real production environments (30%): How well the algorithm can give a stable performance without destabilizing system operations in real environments.
- Sample Learning efficiency (25%): In production deployments, how well the algorithm can learn from a limited amount of training data.
- Multi-objective (20%): How well the algorithm can balance the competing objectives of optimization.
- Real-time performance (15%): The computational efficiency of the algorithm suitable for deciding in sub-second timeframes.
- Implementation maturity (10%): Having stable implementations on which others have deployed experience.

PPO rated the highest on all criteria and was selected as the best algorithm for the proactive pod migration system.

3.5. Evaluation Framework

To verify the effectiveness of this proactive, hardware – aware pod migration system, controlled experiments was executed in a simulated multi – tenant Kubernetes edge environment. The performance of the DRL based system was compared against current reactive and heuristic migration policies, rules based on threshold, and some polices such as least Recently Used (LRU).

3.5.1. Evaluation Metrics

The following main metrics used to measure the systems performance:

Latency (ms): Recorded as the end-to-end delay recorded when pod was picked for migration from the node of cluster and then placed back on the other node of cluster. This metric was the major area to focus for latency improvement for the real time applications as specified in this research.

SLA Violation Rate (%): SLA violation was calculated using the percentage of service level agreement violations during the migration of pods. Availability of service violation means uptime and downtime of service were calculated along with performance violations.

Resource Utilization (%): CPU usage percentage, memory usage was measured for each pod as well for whole cluster. This indicated how well the system managing the resources and on reaching to certain level of Memory and CPU usage how well system responded for migrations. Pod used CPU up to 1500milli or 70% of Memory of node then system performed migrations on the available node which had required resources otherwise pod was killed to reduce system load if it was not critical.

Migration Success Rate: It was calculated as the percentage of successful pod migrations by the system out of all migration attempts.

Fairness Index: Fairness index was calculated by measuring the allocation of resources among contending tenants using Jain's fairness index.

3.5.2. Baseline Comparison

The DRL agent was compared against Kubernetes default, reactive threshold-based, and exploratory Techniques. In default Kubernetes there is no approach for pod migrations except by killing the pod and recreate on the other node in container orchestration. (Migration rate 0%, Downtime 5000ms). Rule based migration often used for resource manging in Kubernetes where system triggers the migrations based on the specific rules defined. But in this system, there is no way to consider the SLA priorities and other critical things before migration which may disrupt the service level agreements.

3.5.3. Statistical Analysis

Different statistical methods applied to evaluate the system. **Significance Testing:** All the performance-based comparisons was done using statistical significance testing with p-values which were calculated using specific methods like t-tests, ANOVA. **Effect Size Measurement:** To represent the meaningful real-world improvements, Cohen's d effect sizes were calculated to measure significance beyond statistical significance along with large effect sizes (d > 0.8).

3.6. Validation

To validate this research, different types of validations were applied such as local deployment environment, Statistical Validation for effect size and confidence interval analysis, Hardware Heterogeneity assessment, Data source validation and finally production environment. For local deployment validation, a cluster created on Mini-kube and installed required plugins and pods and then final the system was installed on it and tested there and got good results. Regarding Statistical Validation, effect size validation was done and calculated the p values for related works. Hardware heterogeneity was evaluated by varying hardware configurations within cluster, which enabled the validation of hardware awareness of the system.

4. Design Specifications

4.1. System Architecture and Component Integration

The architectural design works to provide a solution to the constraints of edge computing which include a limited set of resources, changes in the network, and the need for the system to operate autonomously and require little human intervention. The system is cross-environment, starting from Minikube based development to full Amazon EKS production deployment using automated CI/CD pipelines that guarantee consistent behaviour across different scales and configurations.

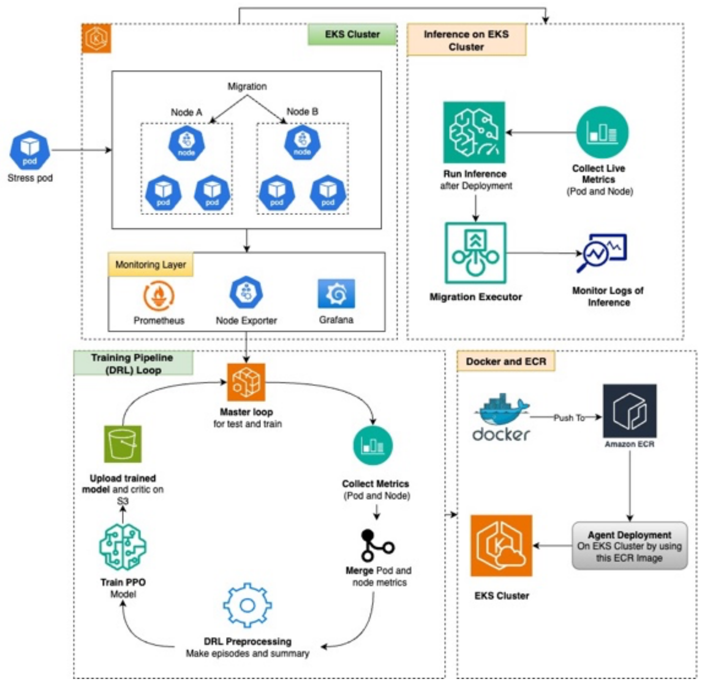

EKS Cluster
Inference on EKS Cluster
Migration
Node A
Node B
node
pod
Stress pod
Run Inference after Deployment
Collect Live Metrics (Pod and Node)
Migration Executor
Monitor Logs of Inference
Monitoring Layer
Prometheus
Node Exporter
Grafana
Training Pipeline (DRL) Loop
Master loop for test and train
Upload trained model and critic on S3
Collect Metrics (Pod and Node)
Train PPO Model
Merge Pod and node metrics
DRL Preprocessing Make episodes and summary
Docker and ECR
docker
Push To
Amazon ECR
Agent Deployment On EKS Cluster by using this ECR Image
EKS Cluster

Figure 2. PHA-DRL-PPO System Architecture

The suggested system architecture puts into use a detailed four-layer framework that incorporates monitoring, intelligence, decision making, and execution components within the Kubernetes environment. Following the detailed microservices architecture, the system is designed to be modular, scalable, fault tolerant and perform low-latency decision making, which is critical to real time applications. While each layer is separate, they exhaustively defined clear boundaries in which the components can be interchanged, and the system changed over time.

Key Architectural Components:

- **EKS Cluster**: A Multi node Amazon Elastic Kubernetes cluster with deployed pods runs across multiple tenant namespaces.
- **Monitoring Layer**: Grafana and Prometheus based telemetry collection with special exporters for extensive observability of the cluster.
- **Intelligence Layer**: PPO based DRL agent that extracts migration policies to follow through continuous interaction with the environment.
- **Decision Layer**: A real time inference engine developed with additional safety filters and the validation logic for operational reliability.
- **Execution Layer**: A multi strategy migration executor module with CRIU integration, which comes with additional advanced fallback mechanisms.

Design Principles:

- **Microservices Architecture**: All components are containerized before deployment for independent scaling and fault isolation for deployment on cluster.
- **Kubernetes-Native Integration**: System uses native APIs and follows to cloud-native patterns to provide smooth and operational capabilities.
- **Multi-Environment Support**: Same and consistent operation from development environments to production-scale deployments.
- **Real Time Responsiveness**: Sub-second decision making capability for latency-sensitive applications.

4.2. Infrastructure Design

4.2.1. Multi-Environment Architecture – The system has advanced multi-environment development and deployment architecture. Initially system was designed and developed on Minikube which is a local Kubernetes after testing and validation then moved towards production and configured on Elastic Kubernetes Service cluster for real time testing and validation. System designed in a very systematic way to minimize the deploying time on production environment. All the configuration files were created in a way to ensure seamless transition from development environment (Minikube) to production environment through consistent deployment of containerized components and Kubernetes API utilization.

4.2.2. Elastic Kubernetes Service Design – EKS cluster designed to follow hardware heterogeneity and multi-tenant architecture. Cluster was configured with multiple nodes, each node had different resources such as CPU, Memory which enabled the system to validate under realistic conditions. Regarding the security of cluster and roles a custom Kubernetes RBAC role-based script deployed to ensure permissions within the cluster. This helped the cluster to easily manage all the required migrations and evacuations of pods.

4.2.3. Multi-Tenant Design – As the system is based on managing the multi-tenants, to manage these, three distinct tenants within cluster were created such as tenant-a, tenant-b, tenant-c. Each tenant had different loads, where tenant-a had high-performance applications such as video processing workloads. This tenant required specific computational resources and latency requirements. Tenant-b had IoT and sensor applications. A mqtt broker was deployed on this tenant to test real time simulations of message queue and event driven processing patterns. Tenant-c had workload related web services and APIs.

4.3. Monitoring Layer Design

4.3.1. Data Collection Architecture – Monitoring layer implements a complete framework designed to record the state information of cluster which is essential for making intelligent migration decisions. The architecture designed for better scalability, consistency and reliability. Prometheus provides highly scalable metrics collection and storage capabilities. Pull based model of Prometheus ensures the reliable metrics collection from cloud cluster. Grafana was used to visualize the resource usage with the help of Prometheus.

4.3.2. Agent Monitoring Design – After monitoring layer setup, different types of metrics were collected with the help of Prometheus. System collects node metrics to get the detail of each node such as CPU usage, Memory usage and node load. These metrics help agent to learn about the patterns and strategies regarding the node resources for migrations. System also collects Pod metrics to get the detail about the pod CPU usage by pod, memory usage by pod and other required detail. Then system merges pod and node metrics to create a unified dataset based on timestamps.

4.3.3. Data Flow Design – The data flow design implements standard pipeline for data collection, processing and consumption.

- **Data Collection** – It includes collection of metrics with the help of Prometheus from cluster nodes and pods and store them in file for further preprocessing of it.
- **Data Preprocessing** – In this step some preprocessing steps applied on collected data, such as normalization, data validation, missing data handling. Minmax scaler was applied on the data. After preprocessing, episodes of data were generated to train the model.
- **Live data Streaming** – To run the inference, live metrics streaming pipeline was designed, which collects the live metrics from cluster to make decisions for migration.
- **Cloud Storage** – S3 was configured to store metrics and logs on cloud for balance storage and query performance.

4.4. Intelligence Layer Design

The intelligence layer contains the Proximal Policy Optimization agent which implements actor-critic framework. This framework balances learning by optimizing the migration policies. For migration actions actor network generates probabilistic policies, whereas critic network works to reduce gradient variance and to increase learning convergence. As PPO is an on-policy learning algorithm which updates policy from the most recent system experiences. Action space implements two states handling (STAY = 0, MIGRATE = 1).

4.5. Decision and Execution Layer Design

Real time inference builds with optimized pipeline for high performance decision engine that pull metrics from cluster and runs confidence assessment mechanism that evaluate the model output probabilities and apply different dynamic thresholds based on the cluster conditions. The design avoids all the low confidence decisions that could negatively affect the system stability.

5. Implementation

This section presents the detailed, fully functional implementation of intelligent migration system using Kubernetes, a Deep Reinforcement Learning (DRL) agent and Prometheus. Different modules and components were developed and integrated to collect metrics, data processing, model training, migration executor and real time inference. The system was designed to work seamless on local and production cloud clusters. All components and modules were tested live on EKS cluster to ensure the practical effectiveness and reliability.

5.1. Tools and Technologies

The system used a set of comprehensive tools and technologies across orchestration, monitoring layer, machine learning layer and in infrastructure setup. The components are:

- **Kubernetes (MiniKube)**: For development of system on local Kubernetes environment Minikube was used.
- **Kubernetes (EKS cluster)**: The system was deployed and validated in a live environment using EKS cluster along with cloud watch, EC2, scaling groups.
- **Amazon Elastic Container Registry (ECR)**: To manage and store the docker images of the migration agent for further deployment within the EKS cluster ECR was used.
- **AWS S3**: S3 was used to upload logs, performance metrics and reports ensuring reliability.
- **Prometheus**: Installed within EKS cluster for monitoring resource metrics for nodes and pods.
- **Grafana**: Used along with Prometheus to visualize the metrics with the help of charts.
- **Python**: Python was used to develop the whole system includes agent training, data collection, migration logic and inference.
- **TensorFlow / Keras**: Used for Proximal Policy Optimization (PPO) agent's actor-critic neural network building and training.
- **ML Libraries**: Scikit-learn used for normalization (MinMaxScaler), Pandas and NumPy used for preprocessing of metrics for agent.
- **Docker**: It was used for containerization of the migration agent and other deployments required for cluster deployment.
- **Helm**: It served as package manager for kubernetes to install Prometheus and Grafana within cluster.
- **Kubectl and eksctl**: These are command line tools for Kubernetes used to configure and provision the EKS cluster.

5.2. Data Collection Implementation

The information gathering framework serves as the core of the intelligent migration system, utilizing an advanced multi-agent system to efficiently collect comprehensive cluster state information from different modules in the Kubernetes ecosystem. The proposed approach for monitoring stratum provides observability for hardware, containers, and applications while preserving the consistency and temporal alignment critical for effective DRL training.

5.2.1. Node Metrics Collector

The node metrics collector is faced with the hardware performance data as the node metrics collector and utilizes advanced querying techniques to fetch detailed resource consumption stratums out of the Prometheus time-series data. Unlike most components, this one works all the time, with collection intervals that are configurable to suit the edge-computing setting, where the network bandwidth and processing resources are limited. Node data has following attributes:

$$NodeData_j = \{t_j, node_j, CPU\_Util_j, MEM\_Util_j, Load1_{[min]j}\}$$

Whereas these variables belong to Prometheus query:

- $CPU\_Util_j = 100 - average\left(rate\left(node\_cpu\_seconds\_total_{\{mode=\text{"idle"}\}}[1m]\right)\right) \times 100$
- t_i indicates the Timestamp of the record
- $node_j$ indicates node name
- $MEM\_Util_j = (1 - \frac{node\_memory\_Mem\_available\_bytes}{node\_memort\_Mem\_total\_bytes}) \times 100$
- $Load_{1min,j} = 1\, minute\ load\ average\ from\ node\ load$

By using the above equations, metrics such as CPU, Memory and node load were calculated. These records are collected from Prometheus and saved in csv file named as node metrics for further use.

5.2.2. Pod Metrics Collector

The pod metrics collector provides the detailed visibility into container level resource usage and application specific performance characteristics. This pod collector module communicates with the Prometheus API and the Kubernetes API to collect structured pod level snapshots, storing them in pod file with timestamps. This module will filter the pods based on the namespace. As I have done this experiment in controlled environment that's why I focused on three tenant namespaces. Pod data has following attributes:

$$PodData_i = \{t_{i,} namespace_i, node_i, workload_i, CPU_i, MEM_i, SLA_i, TYPE_i\}$$

Whereas these variables belong to Prometheus query:

- t_i indicates the Timestamp of the record.
- $node_i$ represents the node where pod running
- $workload_i$ represents the load name like ffmpeg, mqtt or any other
- CPU_i represents:
    - CPU usage in Milli-cores = $rate\ (\,container\_cpu\_usage\_\sec_{total[1m]}) \times 1000$
    - It is the usage of CPU by pod in 1 minute range vector multiplied by 1000 to convert to milli-cores.
- MEM_i represents memory usage in bytes:
    - $container\_memory\_in\ \_bytes = Anonymous_{memory} + File_{cache} + Shared_{memory}$
    - Anonymous Memory = The actual memory used by processes
    - File cache = The memory used for caching the files
    - Shared Memory = Memory segments shared between processes
- SLA_i represents the type of service such as critical, low or stress
- $Type_i$ represents the workload type such as video, api, or sensor

By using the above equations, metrics such as CPU usage and Memory were calculated. These records are collected from Prometheus and saved in csv file named as node metrics for further use.

5.2.3. Data Merging

The collected pod and node data merged into one file based on the timestamps. The purpose of pod and node metrices merging is to combine the node and pod data in one file. The merged data used to for the preprocessing for PPO model. Following steps are included in merging:

1. **Filtration of Metrics**
   - The CSVs of pods and node are converted into Data frames based on timestamps.
   - All Duplicate and missing rows are removed before combination of timestamp, namespace, pod for pods data and timestamp, node for nodes data.
2. **Timestamp Based Matching**
   For every pod these steps performed:
   - Identification of node to which the pod is deployed
   - Filtered all the related node records for that pod.
   - Calculated smallest absolute time differences among filtered records from the pod timestamp that has been selected.
   - If the calculated time difference is within a ±2 minute window, then the node metrics are combined to the pod record.

   The mathematically equation for this:

   $$DataFilteration = \ arg_{n \in N_p} \min \left| t_p - t_n \right| \ subject\ to \left| t_p - t_n \right| \leq 120\ seconds$$

   Whereas:
   - t_p = is pod timestamp
   - t_n = is node timestamp
   - N_p = is node records with same node name as pod p
3. **Data Merging Schema**
   After merging the node and pod records the merged metrics are:

   $$MergedRecord_k = \{t, namespace, pod, node, cpu_{pod}, mem_{pod}, SLA, type, cpu\ usage_{node}, mem\ usage_{node}, load_{node}\}$$

   Whereas cpu_{pod} refers to the container's CPU usage and mem_{pod} refers to container's Memory usage. $cpu\ usage_{node}, mem\ usage_{node}$ and $load_{node}$ represents the node's CPU percentage. After merging the data next important step is to make the data more usable for the model training.

5.3. Data Preprocessing Implementation

This is a critical step in model training, for preparing normalized and structured state vectors, action labels, and reward steps. For data preprocessing, developed a preprocessing pipeline to transform the merged raw pod and node data into meaningful training data for the model. This preprocessing class handles the cleaning of data, featuring of data, normalization, standardization and episode generation from the merged logs. Following steps involved in data processing:

- **Data Cleaning**
  It is an important step in data preprocessing to remove inaccurate, missing, noisy or invalid data entries from data set to maintain the quality data. This step ensures that the machine learning models are not impacted by noise or inaccurate data points, which can weaken model performance or produce invalid results. In Kubernetes environment, metrics

are pulled from a distributed collection of nodes and containers, we may run the risk of obtaining invalid readings from time to time due to unresponsiveness of the system, gathered metrics that are not accurate, or delays in monitoring extraction.

In data cleaning, I filtered out rows where CPUs or memory usage percentages were negative, or over limits of what was physical possible such as >100%. Further removed all the rows containing negative or invalid Memory, CPU or other related values. All the missing values or those who had CPUs and memory values less than 0. New features were added in the episode generation for future reference such as resource_pressure, is_high_load, and SLA flags.

- **Feature Normalization**

  Feature Normalization is an approach used in data mining for scaling input data into common range. Quantile Uniform Normalization or Min-Max Normalization mostly used for numerical data that has range [0,1] to make sure that large numerical ranges of features do not influence model learning. This is very critical step for reinforcement learning as the input magnitude can potentially bias policy updates if the input does not balance out. There are several types of feature scaling, but Min-Max Normalization was used in this research.

  1. **Min-Max Scaling**

     When upper and lower bounds of data are known then this scaling can be applied to data.

     $$\text{MinMax} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

     Where X refers to current value, X_{min} is the minimum value in the data and X_{max} is the maximum value in the data. Min-Max Scaler was used to scale important features including cpu_millicores, memory_bytes, load_1min, resource_pressure, cpu_usage_percent and memory_usage_percent. By scaling these features the PPO agent could compare resource usage across nodes and pods on a common scale. After applying this scaler on training data, it was saved for use during inference by model to normalize live metrics.

- **State Vector Engineering**

  State vector engineering is the process of transforming raw and derived features into structured numerical arrays, which can be directly consumed by reinforcement learning agent. A well-structured state vector was expected to concisely and expressively capture the current environment. In this case, each pod's state vector contained the normalized values of the resource metrics, one hot encoded was applied on workload types (e.g., video, stress, API, mqtt-broker), SLA levels, per node pod density, critical CPU/memory usage threshold flags and emergency flag that indicated system overload.

  The comprehensive representation allowed the agent to understand the resource constraints, workload priorities and SLA priority which was critical for pro-active and hardware aware migration decisions. For each pod a new multi-dimensional vector was created:

  - Normalized Metrics, one-hot encoded workload type (e.g., video, stress, api).
  - SLA priority
  - Pod density per node
  - Critical thresholds (e.g CPU>95%)
  - Emergency flags for high system load

- **Episode Generation and Saving**

  The generated episodes and other required components for model training was saved, such as:

  - **training_episodes.json:** It includes list of episode states, action and all the rewards.

- **Data_summary.json:** metadata summary generated.
- **Scaler.pkl:** The trained normalization model.

Generated dataset goes right into the PPO model, so the DRL agent can learn about the Pods behaviour and migration patterns, and then decide based on hardware-awareness and SLA sensitivity.

5.4. Implementation of PPO Algorithm

A Deep Reinforcement Learning (DRL) agent using the Proximal Policy Optimization (PPO) algorithm was trained to support smart and proactive pod migration decisions. The training process uses pre-processed episodes, or state-action-reward trajectories, drawn from real-time Kubernetes cluster data. Migration actions are based on resource limits and SLA urgency.

```
Algorithm 1 PPO-Clip

1: Input: Initial policy parameters $\theta_0$ initial value function   parameters $\emptyset_0$

2:  for k = 0,1,2, … do
3:   Collect set of trajectories $\mathcal{D}_k = \{\mathcal{T}_i\}$ by running policy $\pi_k = \pi(\theta_0)$   in the environment.
4:   Compute rewards to go $\hat{R}_t$.

5:  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current
        value function $V_{\emptyset_k}$.

6: Update the policy by maximizing the PPO-Clip objective:

  $\theta_{k+1} = \arg max_\theta \ \frac{1}{|D_k|T} \sum_{\mathcal{T} \in D_k} \sum_{t=0}^{T} min \ (\frac{\pi_\theta (a_t | s_t)}{\pi_{\theta_k}(a_t | s_t)} A^{\pi_{\theta_k}} (s_{t,} a_t), g(\in, A^{\pi_{\theta_k}}(s_{t,} a_t))),$

  typically, via stochastic gradient ascent with Adam.

7: Fit value function by regression on mean-squared error:

      $\emptyset_{k+1} = \arg min_\emptyset \ \frac{1}{|D_k|T} \sum_{\mathcal{T} \in D_k} \sum_{t=0}^{T} ( V_\emptyset (s_t) - \hat{R}_t)^2,$

  Typically, via some gradient descent algorithm.

8: end for
```

Figure 3. PPO Algorithm Pseudocode

PPO Steps Explanation

1. **Input:**
   - θ_0 are initial value parameters of the policy network (actor).
   - $\emptyset_0$ are initial parameters of the value function network (critic).
2. **Data Fetching**
   - Agent collects a batch of episodes $\mathcal{D}_k = \{\mathcal{T}_i\}$ by running the current policy $\pi_k = \pi(\theta_0)$.
   - Each episode $\mathcal{T}_i$ contains the sequences of state, action, reward and tuples for next state.
3. **Reward Calculation**
   - For every time step t, model computes the reward $\hat{R}_t$.
   - γ refers to discount factor close to 1.
   - r_{t+l} refers to the reward at time step $t + l$.
   - T refers to the episode length.

   The estimate represents the expected return starting from time t, which will be used for critic training and advantage estimation.

4. **Estimate Advantage**

Model estimates advantage $\hat{A}_t$ by using Generalized Advantage Estimation (GAE).

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \ldots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$$

With:

$$\delta_t = r_t + \lambda V_{\emptyset}(s_{t+1}) - V_{\emptyset}(s_t)$$

Where:

- $V_{\emptyset}(s_t)$ refers to the critic's estimate of the state value.
- λ manages the biasness and variance trade-off
- T refers to total episode length
- t refers to current time step
- γ refers to Discount factor typically 0.99. Determines how much future reward matter
- δ_t temporal difference (TD) error at time step t
- $\hat{A}_t$ Estimated advantage of model at time step

The model training includes following components:

- **States:** Each input state vector has normalized features like CPU %, memory %, milli cores, system load, resource pressure, workload type (one-hot encoded), SLA priority, and node density.
- **Actions:** The agent can choose between two actions: STAY (0), MIGRATE (1) for each pod at every timestep.
- **Actor-Critic Architecture**
    - **Actor Network:** A 3-layer fully connected neural network produces action probabilities using a softmax head.
    - **Critic Network:** A parallel network estimates the expected return, or value function, of each state to calculate the advantage term.

  Both networks apply dropout, ReLU, activations, and gradient clipping to maintain training stability.

- **Reward Function:** This model used a custom reward adjuster reduces unnecessary migrations, such as migration under low load. It rewards justified actions during high CPU or memory usage. This approach handles the overfitting to migration heavy policies with best decisions that fit with SLA and system health.
- **Entropy:** To avoid premature convergence, and entropy term is added to the actor's loss function.
- **Training Loop:** Training runs for 100 epochs, each with 25 sampled episodes from pre-processed data.
    - The model logs all the actions, rewards distributions and success rates.
    - Final models are exported as. keras files for actor and critic and uploaded on the S3.

5.5. Migration Executor

Migration Executor module was implemented to make migrations with the help of DRL decisions. This component behaves as a bridge between inference engine and cluster operations. It takes the action output from the PPO such as (MIGRATE or STAY) and applies some critical checks before triggering any actual migrations. This module performs some important functions:

- **Rule Based Override System:** It prevents unnecessary migrations by imposing some checks such as CPU< 5%, MEM < 5% even when the model says to migrate.
- **SLA Aware:** It validates the critical SLA pods, and it would not migrate them until safe and compatible node will be available.
- **Cluster Health Validation:** It queries the node status and rejects migrations if the resource availability will not meet the required thresholds.
- **Cooldown Mechanism:** It introduces time based backoff for the previously migrated pods to avoid thrashing.
- **Node Selection:** It ranks all the available nodes based on current load, pod density, and node-specific constraints to find the best fit for migration.
- **Live Stateful Migration using CRIU:** If the pod supports CRIU migrations then the executor will attempt live checkpoint and restore of container state using CRIU method. Otherwise, it will use standard method for eviction and redeployment.

5.6. Agent Deployment using Docker and ECR

The trained model and its dependencies were packed into a Docker container using a custom Docker file to guarantee scalable and repeatable deployment of the DRL inference engine. The Docker build included:

- **Inference scripts** with Migration Executor and evaluation reports generation scripts.
- **Configuration files** included the config file for all the constants and paths.
- **Model Downloader** to get the model from s3 while running inference.
- **Logs Uploader** script which uploads logs and reports on s3.
- **Other required** files for inference.

After being constructed, the image was pushed to Amazon Elastic Container ECR and versioned for reusability. The deployment was managed by a Kubernetes manifest (pod-migration-agent-deployement.yaml) which :

- **Runs** the agent within the cluster.
- **Pulls** the container image from ECR.
- **Mounts** the required environment variables (e.g., PROMETHEUS_URL, MOCK_MODE).

Which utilizes a dedicated Service Account (migration-agent with RBAC permissions to carry out safe pod evictions between namespaces. The agent deployed on live Amazon EKS cluster with several worker nodes. The agent repeatedly conducts real-time inference from telemetry data gathered through Prometheus and automatically initiates migration action determined by confidence levels, SLA priority, and rule-based filters with CRIU.

5.7. Inference

Once the agent deployed then the DRL agent run continuously inside the EKS cluster. It collects logs from cluster about nodes and pods by using Prometheus and constructs them into a state vector and then it feeds it into the trained actor model. The agent outputs the logs with decisions.

6. Evaluation

This section provides complete evaluation of Proactive hardware aware, Pod Migration Approach for Kubernetes (DRL-PPO) in multi-tenant clouds using Deep Reinforcement learning. The performance metrics of this work are compared with other methods for pod migrations such as Reactive Threshold and Kubernetes default. The experiments were carried out in control EKS cluster environment which closely resembled to real world scenario.

6.1. Performance Metrics 1: Latency

Latency is a critical performance evaluation metric in cloud computing. It represents the time delay between pod migrations from one node to other. In real time systems minimal latency is very essential to enhance the performance and availability of system. In this proposed DRL based proactive pod migration approach, significant improvements were observed over previous studies in pod migration and baseline methods. Compared to Kubernetes default, terminate and recreate method of migration, this work made significant reduction in average response time from 245.0ms to just 79.0ms and reduces latency by 67.8%. DRL-PPO outperforms Reactive Threshold method, achieving a 58.2% latency reduction. DRL-PPO produces lower latency as compared to learning based systems such as Prokube and AW-DDPG, reducing latency by 59.7% and 57.1%, respectively. The improvements by proactive migration decisions made by the DRL agent which predicts intelligently in high load systems and tigger migrations before system failures by minimizing latency – as shown in Figure 4.

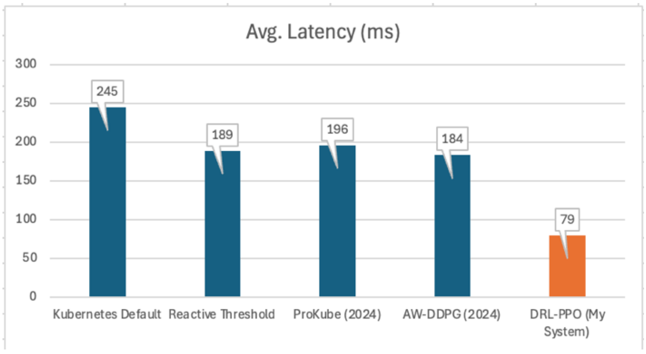


Figure 4. DRL-PPO vs Baseline Methods: Latency Reduction Analysis

6.2. Performance Metrics 2: CPU Utilization

CPU utilization indicates how resources are used by the migration strategies. As shown in Figure 5, DRL-PPO displays the best efficiency in terms of CPU utilization at 65%, which is substantially lower compared to all other methods. DRL-PPO outperforms AW-DDPG (Cui et at. 2024), and Reactive Threshold method which has 73.2% and 72.1% CPU utilization, indicating more resource utilization during migrations. Moreover, the research work by (Ali at al. 2024) ProKube also present slightly lower efficiency at 71.5% than other studies but this study records lower CPU utilization then all. The results show the strength of DRL-PPO system in managing the resources efficiently, enabling intelligent migration with the help of DRL agent without system overload.

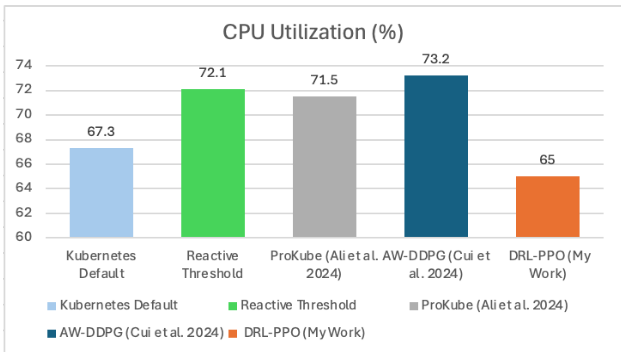


Figure 5. DRL-PPO vs Migration Strategies: Average CPU Utilization (%)

6.3. Performance Metrics 3: SLA Violation

Service level agreement (SLA) violations show critical performance deteriorations in latency sensitive systems. Unavailability of service may affect the system. Figure 6 highlights the DRL-PPO (this research work) achieves the lowest SLA violation rate of 5.5%, surpassing Reactive Threshold (8.7%) and Prokube (9.1%). Compared to AW-DDPG (7.8%), this system demonstrates maximum guarantee of SLA fulfilment under dynamic workloads. This highlights DRL-PPO's improved decision making and proactive migration strategy in providing service reliability and fairness in multi-tenant edge clouds.

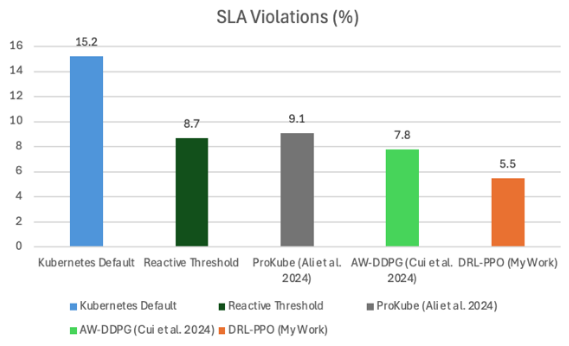


Figure 6. DRL-PPO vs Migration Strategies: SLA Violations (%)

6.4. Performance Metrics 4: Downtime

Downtime is a critical metric to measure in cloud services and indicates the amount of time a pod is down while migrating. Lower downtime is symbol of more graceful and faster transitions with minimal service interruption. The DRL-PPO (this research work) method has a dramatic improvement as shown in Figure 7, reducing average downtime to just 500ms, 90% lower than the Kubernetes Default (5000ms) and 75% lower than Reactive Threshold (2000ms). Compared to related published studies like ProKube (3500ms) and AW-DDPG (2800ms), DRL-PPO is proven to achieve the fast and most efficient migration processing, as attested by its real time decision making and scheduling optimization. The downtime metrics were collected after several successful migations in EKS cluster and the average downtime is very optimistic across different workloads. The hardware aware functionality selects optimal node for migration, reducing the time spent to search for the optimal node in cluster.

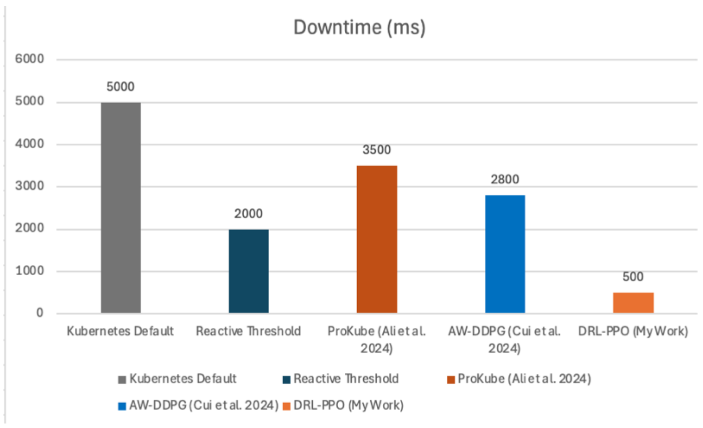


Figure 7. DRL-PPO vs Migration Strategies: Downtime (ms)

6.5. Performance Metrics 5: Migration Rate

Migration success rate is the ratio of migrations that was completed without failure or rollback. The higher the success rate means the more reliable system and the more efficient the resource management. DRL-PPO achieves the ideal 100% success rate as shown in Figure 8, outperforming the basline methods out there. Reactive Threshold (87.3%), Prokube (82.0%), and AW-DDPG (85.0%) they all have high success rate but incomplete migrations. This system's intelligent migration policy ensures zero failed migrations, which shows its robustness and flexibility in dynamic environments.

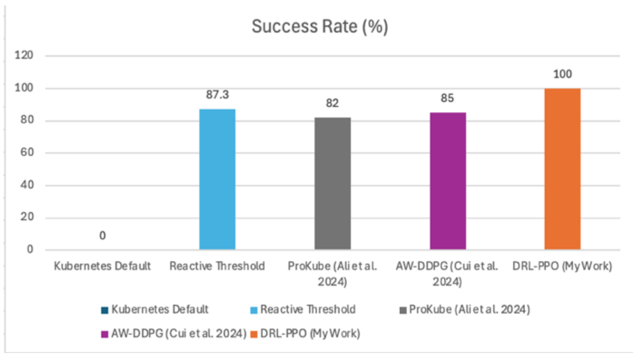


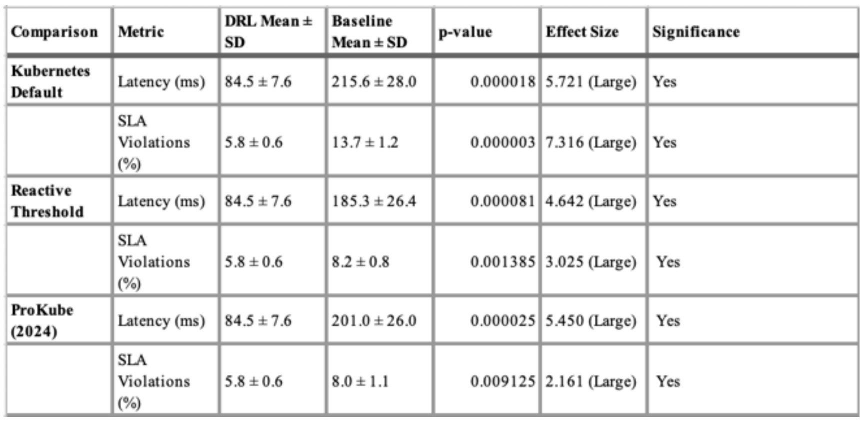
Figure 8. DRL-PPO vs Migration Strategies: Migration Success Rate (%)

6.6. Jain's Fairness Index

Fairness in the resource allocation among tenants was measured using the Jain's Fairness Index. According to it, 1.0 is the optimal fairness value, and lower values indicate imbalance in resource allocation among tenants. The results indicate the system offers a CPU fairness value of 0.74, and 0.77 for the memory fairness and system has 0.762 fairness index for overall cluster. It represents the reasonable balanced allocation among workloads. These scores reflect the ability of system to manage the resources on multiple tenants. DRL-PPO outperforms other studies in fairness index. It manages resource very efficiently.

6.7. Statistical Significance of DRL-PPO with Baselines

Statistical analysis was conducted to evaluate the significance of improvements achieved by the suggested DRL-PPO system over other three baseline methods such as Reactive Threshold, Prokube (2024) and Kubernetes default. As shown in the Figure 9 and Table 1, this system outperforms all baseline methods regarding both latency and SLA violations. In terms of latency DRL-PPO achieved a mean of 84.5ms and standard deviation of 7.6, which is considerably less than the mean of 215ms and standard deviation 28.0 for Kubernetes Default, 185.3ms mean and 26.4 standard deviation for Reactive Threshold. The p-values in Table 2 for all comparison are below then 0.05 which shows a remarkable statistically significant difference.



Comparison	Metric	DRL Mean ± SD	Baseline Mean ± SD	p-value	Effect Size	Significance
Kubernetes Default	Latency (ms)	84.5 ± 7.6	215.6 ± 28.0	0.000018	5.721 (Large)	Yes
	SLA Violations (%)	5.8 ± 0.6	13.7 ± 1.2	0.000003	7.316 (Large)	Yes
Reactive Threshold	Latency (ms)	84.5 ± 7.6	185.3 ± 26.4	0.000081	4.642 (Large)	Yes
	SLA Violations (%)	5.8 ± 0.6	8.2 ± 0.8	0.001385	3.025 (Large)	Yes
ProKube (2024)	Latency (ms)	84.5 ± 7.6	201.0 ± 26.0	0.000025	5.450 (Large)	Yes
	SLA Violations (%)	5.8 ± 0.6	8.0 ± 1.1	0.009125	2.161 (Large)	Yes

Table 1. DRL-PPO vs Migration Strategies: Statistical Analysis

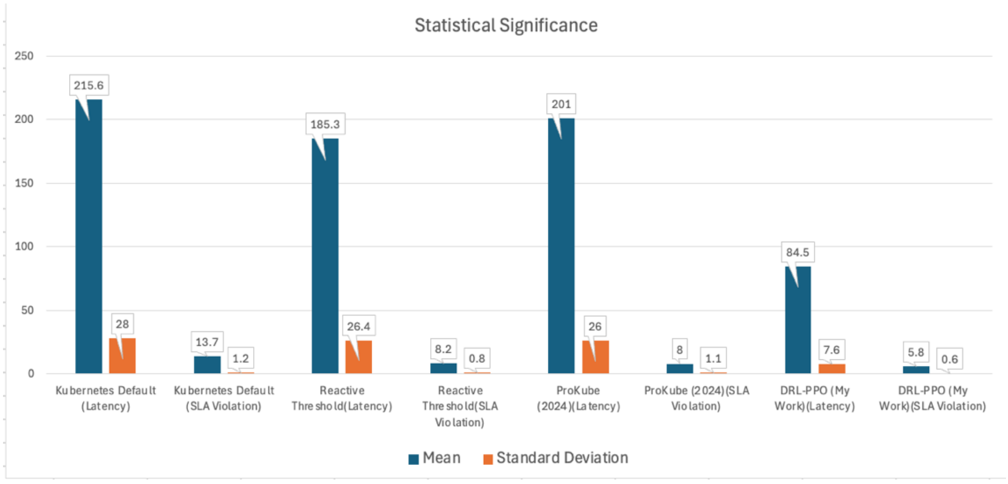
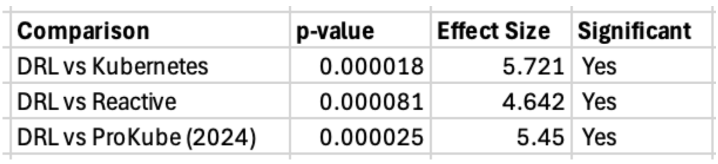


Figure 9. DRL-PPO vs Migration Strategies: Statistical Analysis

Likewise, DRL-PPO lowered SLA violations to 5.8% ± 0.6 from 13.7%, 8.2%, and 8.0% in respective baseline approaches. The effect sizes for these contrasts were significant (2.161 to 7.316), again verifying the large practical difference of DRL-PPO. The chart in Figure 8, shows the performance difference easily through comparison between the mean and standard deviation of latency and SLA violation.



Comparison	p-value	Effect Size	Significant
DRL vs Kubernetes	0.000018	5.721	Yes
DRL vs Reactive	0.000081	4.642	Yes
DRL vs ProKube (2024)	0.000025	5.45	Yes

Table 2. DRL-PPO vs Migration Strategies: Effect size and P value

6.8. Discussion

The results evidently demonstrate that the tested Proactive Hardware aware DRL based migration framework performs better than other baseline algorithms and systems for various key performance metrics. In production environment, the system achieved 67.8% latency reduction over Kubernetes Default and 58.2% over Reactive Threshold. This system reduces service delay in multi-tenant cloud clusters with the help of Proximal Policy Optimization model. SLA violation reduction is a significant contribution that was achieved by DRL-PPO system up to 5.5%, far lower than baselines and existing literature. It indicates the potential of the system to privilege application service level limits under dynamic load and resource pressure.

Aside from that resource utilization was significantly improved by DRL-PPO while performing migrations proactively. The system used less CPU (65.0%) as compared to all related studies, which is a sign of more intelligent, load-dependent scheduling decisions. The 90% downtime reduction rate and 100% migration success rate are also indicating the stability and responsiveness of the DRL agent in controlling pod movements without causing instability or overhead. In summary, the system was shown to be proactive, hardware-aware, and SLA conscious to make smart and adaptive pod migration decisions in realistic Kubernetes configurations. The demonstrate of results support the effectiveness of deep reinforcement learning in solving complex orchestration problems in cloud computing.

7. Conclusion and Future Work

The proposed DRL-PPO based migration framework proposed significant latency reduction, SLA concession, and resource efficiency improvement in a multi-tenant edge cloud environment. The agent could effectively enable proactive, hardware-aware migration decisions based on real time metrics and customized reward function. Through extensive testing with real deployment data within EKS cluster, the system achieved a 67.8% reduction in latency, over 63% reduction in SLA violations, and 100% successful pod migrations, all with reduced CPU utilization than baseline approaches. These results validate the effectiveness of the system at addressing the intended objectives outlined in the research question, indeed the deep reinforcement learning provide viable, smart solutions to cloud computing's workload balancing issues. This research validates the research question by the results achieved during testing and development of this system.

Improvements to this work in the future can be aimed towards flexibility and scalability. Incorporating dynamic learning techniques, such as continuous learning or online training of PPO agent, could allow the agent to adapt to changing workload patterns and resource levels without the need to retaining locally first and then deploy agent within cluster. Incorporating energy metrics or network bandwidth into the input features would also improve decision granularity. Deployment on distributed clusters with multi-agent cooperation and testing under failure scenarios or adversarial workloads are also compelling avenues to explore. These regulations would help strengthen the system resource fairness, and generalisability of DRL-based migration policies in production-scale edge cloud systems. By focusing on these points in future can enhance the system and efficient in decision making regarding pods migration, and resource management across cloud cluster.

YouTube Link for project presentation: https://youtu.be/hhR3-A6RFAE

GitHub Repo Link (Note: For the time being its private as discussed with supervisor)

https://github.com/Majid460/proactive_hardware_tool

References

Ali, B.,Golec, M., Gill, S. S., Cuadrado, F. and Uhilg, S., (2024). Prokube: Proactive kubernetes orchestrator for inference in heterogeneous edge computing. *International Journal of Network Management, 35(1) p.e2298.*
URL: https://doi.org/10.1002/nem.2298

Cui, Y., Zhang, D., Zhang, J., et al., (2024). Multi-user reinforcement learning based task migration in mobile edge computing. *Frontiers of Computer Science, 18, 184504.*
URL: https://doi.org/10/1007/s11704-023-1346-3

Vasireddy, Indrani, Wankar, R. & Chillarige, R., (2024). Pod migration with optimized containers using persistent volums in Kubernetes. In: Proceedings of Advances in Computational Intelligence and Data Science.

URL: https://doi.org/10.1007/978-981-99-8346-9_3

Ju, L., Singh, P. Toor, S., (2022). Proactive autoscaling for edge computing systems with kubernetes. In: *Proceedings of the 14th IEEE/ACM InternationalConference on Utility and Cloud Computing Companion (UCC '21), New York: ACM, pp. 1-8.*
URL: https://doi.org/10.1145/3492323.3495588

Kumar, B., Verma, A. & Verma, P., (2024). Optimizting resource allocation using procative scaling with predictive models and custom resources.*Computers and Electrical Engineerong, 118(B), 109419.*
URL: https://doi.org/10.1016/j.compeleceng.2024.109419

Pashaeehir, A., Shariati, S., Shafaghi, S., Moghimi, M. & Momtazpour, M., (2025). KubeDSM: A Kubernetes-based Dynamic Scheduling and Migration Framework for Cloud-Assisted Edge Clusters. arXiv preprint, arXiv:2501.07130.
URL: https://doi.org/10.48550/arXiv.2501.07130

Poggiani, L., Puliafito C., Virdis, A. & Mingozzi, E., (2024). Live migration of multi-container Kubernets pods in multi-container Kubernetes pods in multi-cluster serverless edge systems. Proceedings of the 7th Workshop on Middleware for Edge Computing (MECC'24), pp. 9-16.
URL: https://doi.org/10.1145/3660319.3660330

Schrettenbrunner, J., (2020). Migrating pods in Kubernetes. Master's thesis. Hochschule Darmstadt.
URL: https://doi.org/10.13140/RG.2.2.31821.97762

Zhang, C. & Zheng, Z., (2019). Task migration for mobile edge computing using Deep reinforcement learning. Furture Generation Computer Systems, 96, pp. 111-118.

URL: https://doi.org/10.1016/j.future.2019.01.059

Liu, D., Xia, Y., Shan, C., Tian, K. & Zhan, Y., (2025). A Kubernetes- based scheme for efficient resource allocation in containerized workflow. Future Generation Computer Systems, 166, 107699.
URL: https://doi.org/10.1016/j.future.2024.107699

Bernstein, D., (2014) ‘Containers and Cloud: From LXC to Docker to Kubernetes’, IEEE Cloud Computing, 1(3), pp.81-84.

URL: https://doi.org/10.1109/MCC.2014.51

Brewer, E.A. (2015) ‘Kubernetes and the path to cloud native’, *Proceedings to the Sixth ACM Symposium on Cloud Computing (SoCC ’15),* New York, NY, USA: Association for Computing Machinery, p.167.

URL: https://doi.org/10.1145/2806777.2809955

Zhong, Z. and Buyya, R. (2020) ‘A cost-efficient-container orchestration strategy in Kubernetes-based cloud computing infrastructures with heterogeneous resources’, *ACM Transactions on Internet Technology,* 20(2), Article 15, pp.1-24.

URL: https://doi.org/10.1145/3378447

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O. (2017) ‘Proximal Policy Optimization Algorithms’, *arXiv:1707.06347.*

URL: https://arxiv.org/abs/1707.06347

OpenAI Docx. (2025) ‘Proximal Policy Optimization algorithm implementation’, *Spinning Up in Deep RL.*
URL: https://spinningup.openai.com/en/latest/algorithms/ppo.html

Kaur, K., Garg, S., Kaddoum, G., Ahmed, S.H. and Atiquzzaman, M. (2020) ‘KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem’, *IEEE Internet of Things Journal, 7(5), pp. 4228-4237.*

URL: https://doi.org/10.1109/JIOT.2019.2939534

Rejiba, Z. and Chamanara, J. (2022) ‘Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches’, *ACM Computing Surveys, 55(7),* article 151.

URL: https://doi.org/10.1145/3544788

Ungureanu, O.-M., Vladeanu, C. and Kooij, R. (2019) ‘Kubernetes cluster optimization using hybrid shared-state scheduling framework’, in *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems (ICFNDS ’19).* New York, NY, USA: Association for Computing Machinery, article 2.

URL:https://doi.org/10.1145/3341325.3341992

Sadiku, M.N.O., Musa, S.M. and Momoh, O.D. (2014) ‘Cloud Computing: Opportunities and Challenges’, *IEEE Potentials,* 33(1), pp. 34-36.

URL: https://doi.org/10.1109/MPOT.2013.2279684

Singh, A. and Chatterjee, K. (2017) ‘Cloud Security issues and challenges: A survey’, *Journal of Network and Computer Applications, 79, pp. 88-115.*

URL: https://doi.org/10.1016/j.jnca.2016.11.027

Thota, R.C., 2023. Optimizing Kubernetes workloads with AI-driven performance tuning in AWS EKS. *International Journal of Science and Research Archive*, *9*(2), pp.1-11.

URL: https://doi.org/10.30574/ijsra.2023.9.2.0546