Learning-Based Anomaly-Tolerant Resource Management in Cloud-Edge Continuum

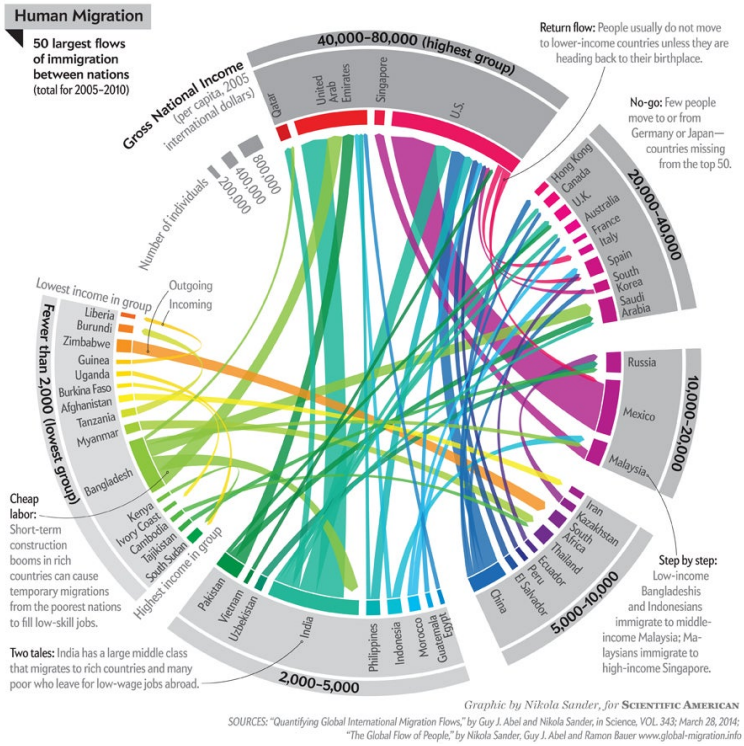
Fill In Your Sub-Title

submitted in partial fulfillment for the degree of master of science

Nils Barring
13983342

Master Information Studies
data science
Faculty of Science
University of Amsterdam

Submitted on 31.07.2023



	UvA Supervisor	External Supervisor
Title, Name	Hongyun Liu	External Supervisor
Affiliation	UvA Supervisor	External Supervisor
Email	h.liu@uva.nl	supervisor@company.nl

Amsterdam Data Science

The Awesome Company ™

ABSTRACT

In this study, we propose a model aimed at simultaneously optimizing job slowdown and detecting anomalies to enhance the efficiency of the cloud-edge continuum. Despite the growing importance of these two topics, the literature surrounding it is scarce and fails to investigate the potential synergy of combining both concepts. To address this gap, we present two Deep Reinforcement Learning (DRL) models: DeepRM_ECO and DeepRM_AD. These models harness the strength of DRL to minimize job slowdown and effectively detect and handle incoming anomalous jobs. Our experimental results demonstrate that both DeepRM_ECO and DeepRM_AD significantly outperform the baseline DeepRM in minimizing the mean job slowdown as a slowdown of 5.4 and 3.2 respectively reach, against DeepRM's mean slowdown of 9.8. Remarkably, DeepRM_ECO has exhibited a high recall in the range of 0.952 to 0.998 for detecting and treating anomalies irrespective of the anomaly rate. In practical terms, our study aim to provide some insights on utilizing DRL methods as a learning based anomaly tolerant resource management solution for optimizing the cloud-edge continuum.

KEYWORDS

Deep reinforcement learning, Job scheduling, anomaly detection, distributed systems, optimization of cloud-edge continuum, fault-tolerance

GITHUB REPOSITORY

https://dagshub.com/n.barring/deeprm_anomaly.git, https://dagshub.com/n.barring/DeepRM_ECO.git

1 INTRODUCTION

The objective of this paper is to provide a method that optimizes the job scheduling process while being able to detect and treat anomalous tasks to reduce the risk of hardware and system dynamics failures.

As we shift towards a data driven society[24], the complexity and volume of tasks executed within the cloud-edge continuum has increased exponentially. As a result of this surge, the need for efficient job scheduling policies becomes primordial. Job scheduling is the process of allocating computational resources to perform a computing task within a system, while ideally aiming to optimize some metrics such as throughput maximization, job slowdown minimization, energy consumption and more[6]. Alongside the challenge of job scheduling, the increased complexity and use of the cloud-edge continuum has exposed these systems to various anomalies such as hardware failure, system dynamic failure or anomalous workload. Thus, this combination of problems could potentially lead to a less reliable system and harder scheduling decisions[11].

Hence, it is imperative for cloud providers to tackle the challenges of efficiently and fairly allocating resources, handle diversity in job requirements as well as deal with anomalies to prevent faults within the system[12].

To address these challenges, several studies have been conducted with the goal of optimizing the job scheduling process [1, 7, 9, 15, 20, 21, 23, 27], or the detection of anomalies[8, 25]. Despite plenty of studies undertaken to identify an optimal scheduling policy in edge or cloud computing, job scheduling is recognized as an NP-Complete problem[26]. Thus, the task increases exponentially in complexity as the problem size gets larger. For this reason, most researchers have now shifted from brute-force techniques towards using a combination of heuristics [1, 21, 23], meta-heuristics [15, 20] and machine learning methods [7, 9, 29, 30] to derive a near-optimal solution within a reasonable timeframe.

To this day adapted versions of traditional heuristics such as SJF[1], FCFS[23] are often the foundation upon which more complex scheduling systems are built. For example, Hadoop YARN uses a combination of scheduling algorithms including FCFS and Fair Scheduler[14]. Another example is Google Borg who also uses a combination of scheduling heuristics including priority preemption[27]. The use of traditional approaches often results in near optimal scheduling, but often fall short when dealing with scheduling tasks because of their deterministic rules and inability to adapt. In contrast, machine learning based approaches have demonstrated potential to address these challenges. For example, deep reinforcement learning (DRL) approaches such as [7, 9] have shown to be well suited for such tasks as it is able to dynamically adapt its policy through interactions with the environment[4, 10]. Mao et al.[9] demonstrated the potential of DRL in their algorithm DeepRM which outperformed popular scheduling heuristics such as Shortest-Job-First (SJF)[1].

Although finding the optimal scheduling policy for the well functioning of cloud systems cannot be understated, the significance of anomaly detection to prevent potential system failure holds equal weight. The importance of fault tolerance is even more emphasized as the demand and scale of cloud systems increases and complexifies[2, 24]. Two main strategies underpin the concept of fault-tolerance as explained by Ledmi et al.[13]: the reactive way which consists of dealing with the anomaly post-occurrence; while the proactive way relies on anticipating potential anomalies and dealing with them before they occur by using techniques such as load balancing and preemptive migrations of tasks. A multitude of researches have been conducted to detect anomalies or outliers, as was shown by Han et al.[8]. For example, Deep Neural Networks have been used to detect anomalies within images and natural language data[8]. Other methods such as support vector machines (SVM), Gradient Boosting or k-nearest neighbors (KNN) have also shown potential in outlier detection[8]. Whilst these algorithms have shown strength in their respective domain, there is no single algorithm that outperforms the others at every task[8]. A promising research on fault tolerance within a cloud system was offered by Tuli et al.[25] who proposed PreGAN. PreGAN reached state of the art status in the context of faults detection and preemptive migration decision within a small scale cloud system.

While impressive results have been reported for fault detection [8, 25] and job scheduling/resource management[7, 9], there still exists a lack of literature regarding the unification of these two distinct yet inherently related concepts under a single algorithm.

Most research done in these domains have investigated these two concepts in isolation, which might have failed to capitalize on the potential synergy of interplaying job scheduling and anomaly detection within cloud systems. We therefore aim to bridge this research gap by implementing an adaptive DRL[4] algorithm that simultaneously handles job scheduling as well as anomaly detection and treatment.

For this, we present DeepRM_AD and DeepRM_ECO, two solutions that builds upon DeepRM[9] by reconciling optimal job scheduling, anomaly detection and treatment while leveraging the adaptive nature of DRL. DeepRM_AD and ECO offers two primary functionalities: Firstly, it utilizes the power of DeepRM to optimize job scheduling. Secondly, it introduces an anomaly detection module integrated into DeepRM which proactively identifies and treats anomalies to solidify the fault-tolerance of the system. Through this research we aim to investigate the following research question: **How can learning-based anomaly-tolerant resource management methods be optimized in the cloud-edge continuum?**.

To answer this research question we will focus on solving three main sub-questions:

- *How can we implement anomaly detection and resource management together or in parallel to optimize cloud systems?*
- *How can anomaly or dynamic-related information be treated as an input of the resource management algorithm to optimize it?*

2 RELATED WORK

2.1 Cloud-edge computing:

With the shifting towards a data driven society, the demand for computational power in our daily basis has skyrocketed. However, the computational capacity at the edge of the network is very limited and often fails to run expensive jobs[2, 24]. For example, training and running inferences on a non optimized large language model such as the Falcon 40B require data center GPUs with at least 85 to 100GB of memory. As such, using local machines to train such algorithms can be slow or even impossible depending on the hardware. For this reason, the idea of cloud-computing emerged, essentially migrating these tedious tasks to a remote location with the required hardware[11, 24].

Ideally, mixing both edge and cloud computing would be the optimal solution as tasks performed on the edge node have lower latency but are smaller, while cloud-computing provides the possibility to perform larger tasks[24]. Therefore, there is a trade-off and it is essential to find a way to optimize the allocation of tasks and the fault-tolerance of both.

2.2 Job Scheduling:

The problem of scheduling has broad implications and have been investigated in many fields. Yu et al.[31] recognize job scheduling as a core concept in areas such as workshop production processes. In the context of scheduling jobs in a cloud-edge computing system, the challenge revolves around ordering jobs or tasks such that a chosen metric is optimized. For example, Akhtar et al.[1] minimized the waiting time for jobs to be scheduled on a CPU using an enhanced version of the shortest job first heuristic. Liu et al.[15] investigated optimal scheduling for priority scheduler and showed that maximizing the processor utilization could be achieved by dynamically assigning job priorities based on their current deadlines. Therefore, scheduling is a broad challenge that is composed of both minimization and maximization problem based on the context and the objectives.

Ullman[26] demonstrated that job scheduling had an NP-Complete nature. As such, the time to find an optimal solution can increase exponentially with the size of the problem. Because of the complexity of the problem, heuristics and approximation algorithms are often used to derive a near optimal solution in reasonable time. In the context of cloud or edge computing systems scheduling is often done by using a combination of fine tuned heuristics such as First-Come-First-Serve (FCFS)[23], Shortest-Job-First (SJF)[1], priority preemption[15] or fair tree algorithm[22]. For example, Lin et al.[14] proposed a performance evaluation of Hadoop YARN job scheduler which uses a mix of capacity scheduler, FCFS heuristic and a fair scheduler. While Google Borg uses a mix of heuristics including priority preemption as discussed by Verma et al.[27]. Therefore, heuristics remain a popular approach for job scheduling in large cloud environment because of their simplicity and modifiable nature.

While these solutions are sufficient, as the complexity of the cloud-edge continuum increases it becomes more relevant to find better and more adaptive methods to tackle this problem[2, 24].

With the recent advent in artificial intelligence (AI) and machine learning (ML) partially driven by the explosion in the volume of available data, new avenues have opened up to deal with the challenge of scheduling[6, 7, 9, 29, 30].

Numerous studies have been conducted to investigate if these newer ML based techniques can outperform traditional methods and adapt better to dynamic and complex environments. For example, Mao et al.[9] proposed DeepRM which reached state of the art status at the time. DeepRM uses DRL and imitation learning to help the agent learn more effectively in the given environment. DeepRM was an innovative concept and considerably outperformed most of the popular heuristics. Nevertheless, it is important to remember that DeepRM achieved these results in a relatively small environment with very little variation in job requirements. Similarly to DeepRM we aim to propose a job scheduling algorithm that harnesses the power of DRL in a wider environment.

2.3 Fault tolerance in systems:

Fault tolerance is a critical requirement for the cloud-edge continuum in order to maintain reliability and availability of services[13].

The complex nature of these distributed systems introduces a multitude of potential points of failure. For example, network disruption or device malfunctions which are exacerbated by the unpredictable and diverse workloads[13, 24]. As a result, the necessity for a robust fault tolerant mechanism becomes primordial to deliver a high quality of service. Such mechanisms should be able to detect and manage faults effectively.

Shi et al.[24] and Kelkar et al.[11] discuss the challenges in edge and cloud computing respectively.

Shi et al.[24] identify some key challenges in edge computing that includes optimization of metrics and service management. For example, latency in edge-computing can be optimized by migrating

expensive computing tasks to cloud systems who support significantly more complex workloads in a shorter amount of time due to the superior computational capability[24]. Whereas the service management includes providing a reliable edge system from a service, system and data point of view.

On the other hand, Kelkar et al.[11] present the main challenges in cloud computing. They identify technical issues, reliability and the evolving nature of the system requirement as key challenges of cloud computing. Technical issues refers to the risk of malfunction of the system potentially causing a denial of access to information and data from the cloud. Reliability is the concept of providing a service with minimum downtimes and slowdowns. Finally, the demand and requirement for cloud computing is never static and cloud systems should be able to continuously evolve to meet these requirements[11].

Ledmi et al.[13] state that a distributed system is fault tolerant if the concepts of availability, reliability, safety and maintainability are respected within the system. They further elaborate that fault tolerance can be addressed in either a reactive way or a proactive way. The reactive way aims to address the fault tolerance when it happens and include methods such as retries and check-pointing. For example, Graham et al.[5] proposed Los Alamos Message Passing Interface (LA-MPI) to detect network related failures in terascale clusters in a reactive manner.

In contrast, the proactive approach aims to address the faults before they occur using techniques such as load-balancing or preemptive migration[13]. Algorithms such as PreGAN[25] utilizes preemptive migrations to proactively prevent hardware failure and data loss.

In some cases fault tolerance is addressed in a hybrid way. Ceph storage[28] is one of these hybrid approaches it leverages reactive fault tolerance by replicating data to other nodes to ensure the safety of data. On the other hand, is employs preemptive strategies such as load balancing of data across the system and continuous monitoring to avoid or minimize the effects of faults on the distributed system.

Just like PreGAN[25], we aim to build a module to proactively address potential faults by preemptively migrating anomalous workload to different devices with better resource capabilities. And integrate the anomaly detection module with a job scheduling algorithm to address potential workload that could lead to a system failure.

2.4 Anomaly detection:

The cloud-edge continuum has become increasingly complex with the growing amount of devices that connect to it[11, 24]. Consequently, the continuum is prone to more anomalies making it less fault-tolerant. Given that data is transmitted all over the system there are more opportunities for errors to occur, leading to a less stable and predictable distributed system for example: network latency, connectivity issues, and data corruption. Kelkar et al.[11] found that the complexity of the system can lead to more faults and errors, which can have a cascading effect throughout the network.

Many studies have been conducted to detect anomalies in different contexts. Han et al.[8] conducted a study investigating different anomalies/outlier detection methods such as neural networks, KNN or SVMs. Using a benchmark they show that while these outlier detection methods can perform great in some contexts, they often fail in very dynamic and unpredictable contexts such as distributed cloud or edge systems.

In the context of distributed systems Tuli et al.[25] proposed PreGAN a novel fault-tolerance model that uses generative adversarial networks (GAN) to detect, diagnose and classify anomalies in real time in an online setting. PreGAN has shown to be fast, efficient and scalable, making it comparable to state of the art for real-time operation on fault detection. It is however important to keep in mind that the experiments were conducted on a Raspberry-PI based edge environment. Therefore, the reported results and performance of PreGAN might differ on a more complex setup.

Like PreGAN we aim to deal with anomalies in real-time by using a scale out approach to migrate anomalous workloads across the system. The scale out approach will be integrated into the scheduling algorithm such that both module run concurrently and synergies together.

2.5 Reinforcement learning and Deep reinforcement learning:

Deep reinforcement learning (DRL) is a sub-field of AI that combines deep learning and reinforcement learning (RL)[4]. Unlike other ML approaches RL algorithms or agents learn to make decisions through exploitation and exploration by interacting with a given environment.

However, traditional RL has many pitfalls as discussed by Dulac et al.[3]. In this paper Dulac recognizes the scarcity of sample, the ethics, safety and the complexity of RL algorithms among other causes as the key challenges in real-world application of RL. For example, RL agents often fall short in their ability to generalize experiences because of the scarcity and complexity of the data. Furthermore, agents can be complex to implement, this is especially true in multi-objectives scenarios where complex reward functions and state observation are needed to solve the problem[3]. These pitfalls make RL methods often computationally infeasible, too complex or unscalable.

To overcome these limitations, Mnih et al.[17] proposed the first DRL algoritm: Deep Q-Network (DQN). DQN was tested on classic Atari 2600 games and performed better than any of the previous algorithms and even better than professional game testers. Mnih et al.[17] attributed the success of DQN to the use of a deep neural networks (DNN) which permitted to learn from high-dimensional state spaces.

Overall, the success of DRL is partially attributed to the use of DNN that permits the generalization of high dimensional inputs. Precisely, the DNN maps the input to the available actions which then serves as a function approximator making it capable of generalizing experiences in unseen states[4].

Several studies have employed DRL in cloud systems or cloud-edge continuum to optimize various metrics.

For example, DRL has been used to tackle the problem of datacenter power management. Liu et al.[16] introduced dynamic power management (DPM), a framework that aim to minimize the power consumption and cost of datacenters while maintaining an acceptable performance degradation. DPM resulted in significant savings in power consumption and energy usage by using an auto-encoder,

an LSTM based workload predictor and a model free RL based power manager.

In the context of workload scheduling a multitude of studies have been undertaken as Zhou et al.[6] highlight in their review of DRL-based methods for cloud computing. For example, Harmony proposed by Bao et al.[30] which uses a DRL framework to minimize interference and maximize performance by placing training jobs optimally in distributed machine learning clusters. Harmony uses a deep learning driven cluster scheduler, state of the art DRL techniques such as actor-critic algorithm, job aware action space exploration and experiment replay. At the time Harmony was able to outperform popular schedulers by 25% in average job completion time. Another study focusing more on the optimization of cloud-edge computing has proposed two resource allocation algorithms, CERAI and CERAU, both based on DRL. These algorithms are proposed by Jianqiao et al.[29] and aim to reduce cost and allocate resources efficiently in both private and public cloud-edge computing systems. Another example previously mentioned is DeepRM presented by Mao et al.[9] which pioneered the use of DRL algorithms for resource management in cloud computing and edge computing. In this paper, we revisit the implementation of DeepRM by scaling out the implementation and adding am anomaly detection module powered by DRL that will run in parallel to the job scheduler such that anomalous jobs are detected and migrated, all while keeping a near optimal job scheduling policy.

A lot of studies have been conducted into treating anomalies or managing resources, however, very few studies have tried to fuse these two concepts.

This study aims to bridge the gap in literature by providing a consistent job scheduling algorithm that is able to take as input anomaly related information and optimize the scheduling by treating anomalies and managing resources simultaneously.

3 PROBLEM DEFINITION

3.1 Reinforcement learning

As our project will be heavily centered around RL, we will give a brief overview of the foundational concepts in RL. A RL problem is traditionally composed of numerous components but the primary ones are the following[4, 10]:

- The agent: The agent is the entity that learns to make decisions in an environment through interactions and rewards received in each state.
- The environment: The representation of the world in which the agent moves. The environment receives as input the agent's current state and actions and outputs the next state with the reward.
- The state: A state s represents a certain position or condition of the agents in their environment at time step t. At each state a different set of actions a are available to the agent. Traditionally, once an action is taken by the agent, the state transitions to the next one: s_{t+1}.
- A set of actions: They are the different options that the agent has to interact with the environment in a given state s. When an action a is taken in state s_t, the state of the environment can transitions to s_{t+1} and a new set of action becomes available. In our case, the state transition is stochastic and follows the Markov property were the state transition probability only depend on the current state of the environment s_t and the action selected a_t by the agent.
- The reward function: The reward function serves to give feedback to the agent after an action a_t to indicate how well the agent performed in the current state. Just like actions, the reward is stochastic and follows the Markov property and therefore only depend on the state and the action selected.
- A value function: The value function is used to estimate the expected cumulative future reward for being in a given state. Many different value functions exist, but the traditional ones are the Bellman Equation[18], and the Q-function[19]. In the case of DeepRM we are using: $E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$ where $\gamma \in (0, 1]$ is the discount factor which allows control of the trade-off between immediate and future rewards of the agent.
- A policy: It is the strategy that the agent will follow to determine the best action to take in the current state. The policy is defined as: $\pi : \pi(s, a) \rightarrow [0, 1]$ where $\pi(s, a)$ represent the probability of taking action a in state s.
- Function approximator: As there can be millions of different actions in an environment, DeepRM uses Deep Neural Network as a function approximator. The logic behind these approximator is that the agent should normally take similar actions for states that are "nearby". The use of DDN as function approximator transforms the nature of the algorithm from a RL to a DRL model.
- Policy gradient method: DeepRM uses gradient descent on the policy parameters to maximize the expected cumulative discounted reward as such:

$$\nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a) Q_{\pi^\theta}(s, a)\right].$$

The expected cumulative discounted reward of choosing an action a in state s when following the policy π_θ is represented by: $Q_{\pi^\theta}(s, a)$. DeepRM uses simple Monte Carlo Method to sample a handful of trajectories to estimate the gradient. Then the agent samples multiple trajectories and use the empirically computed cumulative discounted reward obtained v_t to estimate $Q_{\pi^\theta}(s, a)$. Finally, it updates the policy parameters using the following gradient descent equation:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

In gradient descent the parameter α represent the step size. In this equation, ∇_θ allows to change the policy parameters to increase or decrease the probability of of selecting a_t at state s_t depending on the direction of ∇_θ. Furthermore, the above equation takes a step in the direction of ∇_θ where the size of the step correlate to the size of the return v_t. So, essentially the policy is updated to increase or decrease the probability of selecting a_t in s_t by using gradient descent.
- The exploration vs exploitation trade-off: This trade-off prevents the agent from exploiting a sub-optimal policy by forcing it to explore new states.

4 METHODOLOGY

4.1 Dataset and exploratory data analysis:

Our objective is leveraging the strength of DRL in a job scheduler such that it can efficiently adapt and treat anomalies in different scenarios. But since DRL models require extensive experience and data to learn effectively and because of the scarcity of cloud computing or edge computing related data, many DRL implementations rely on synthetic datasets to train and find the optimal policy. Therefore, our agent will be trained on a dataset generated by DeepRM. While this dataset might appear unrealistic because of its synthetic nature, it provides several advantages for our research.

- First, it allows us to simulate a wide range of scenarios by adjusting parameters. To that end our experiments can be more flexible and the performance of the model can be tested in various environments.
- Secondly, synthetic datasets enable us to generate a large amount of data which is primordial to effectively train DRL models[3].
- Finally, the data generating feature offers sophisticated modeling capabilities, making it flexible and easy to generate data that closely mirrors real-world scenarios.

To incorporate anomalous data in the dataset we changed the data generation module to produce "anomalous" jobs with either an abnormal length of execution or a resource requirement higher than normal jobs. The generated jobs can either be small, big or anomalous. Their generation is randomized but also controlled as chances are low for an anomalous jobs to be created while small and big jobs have a higher chance of being generated. But these generation rates can be changed according to the scenario we want to create as seen in pseudo-code8.

Data features: In our implementations of DeepRM, a job data is composed of six main features;

- The job id: We denote the job id as j_i, where $j \in J$ is the set of all job, and i is the unique id of job j.
- The job resource requirement vector: We represent the resource requirement of a job j as a vector:

$$\vec{r_j}[[r1_{j1}, r2_{j1}], ..., [r1_{jn}, r2_{jn}]]$$

  where r represents the amount per type of resource required by the job j. In our scenarios, a normal job will have a demand for both resources in the range of 1 to 10:

$$[r_n = \{r \in \mathbb{N} \mid 1 \leq r \leq 10\}]$$

  . While for an anomalous workload, one of its required resources will be in the numerical range of 11 to 15:

$$[r_a = \{r \in \mathbb{N} \mid 11 \leq r \leq 15\}]$$

  where r_n represent normal jobs and r_a anomalous ones.
- The job length: The length of job j is represented as j_L. A normal job will have a length in the range of 1 to 18:

$$[L_n = \{L \in \mathbb{N} \mid 1 \leq L \leq 18\}]$$

  and an anomalous task will have a length in the range of 19 to 29:

$$[L_a = \{L \in \mathbb{N} \mid 19 \leq L \leq 29\}]$$

  . L_n represent a normal length and L_a represent an anomalous length.
- The job enter time: The enter time of job j is denoted as j_E.
- The job start time: The start time of job j is denoted as j_S.
- The job finish time: The finish time of job j is denoted as j_F.

When finding the optimal policy, the job resource vector $\vec{r_j}$ and job length j_L are the primary features to take into consideration for scheduling jobs. The enter time j_E, start time j_S and finish time j_F of jobs are used to calculate metrics to optimize such as the job slowdown, work completed or the unfinished workload.

Dataset exploratory data analysis: The distribution of the dataset vary based on the experiments and scenarios. For example, with the same parameters as shown in the pseudo-code8 we get roughly 5% of anomalous jobs based on their resource demand as seen in figure 1. While for the rate of anomalous jobs based on length we get around 8% as seen in figure2.

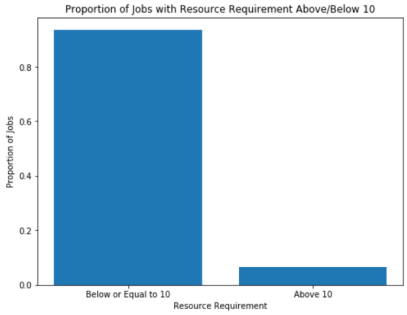


Figure 1: Proportion of normal and anomalous jobs based on resource needed

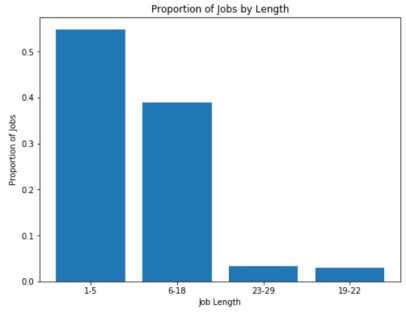


Figure 2: Proportion of normal and anomalous jobs based on length

4.2 Model used:

Since DeepRM is a fairly recent open source DRL model that provides a complete, simple yet effective code base and because of the high performance of DeepRM demonstrated by Mao et al.[9], we will use the original source code of DeepRM as a framework for developing our project. In order to test different scenarios, assumptions and constraints, we propose two different variations of DeepRM, DeepRM_AD and DeepRM_ECO.

4.2.1 DeepRM_AD: . DeepRM_AD for anomaly detection, aims to test the strength of RL for anomaly detection when limited constraints in regards to the maximum resource available for allocation are in place.

With DeepRM_AD jobs can be allocated to any instance regardless of if they are anomalous or not. In that case the instances have a maximum resource capacity of 15, meaning that in practice any jobs can be allocated even if they have an anomalous demand for resources r_a.

The RL agent therefore needs to be able to distinguish anomalous jobs and migrate them to a separate instance without the help of fixed constraints. As a result we are simulating a scenario in which we have one edge instance and many cloud instances all capable of running any incoming jobs but preferably high demanding jobs should be migrated to the cloud to prevent slowdowns caused by high resource demand r_a or long execution time L_a. Overall, with this model we test the performance of RL to detect anomalies when limited to no constraints are in place.

Architecture: DeepRM_AD's environment provides five components with their own actions and properties as seen in diagram 4.

- The backlog holds all the jobs waiting to be placed in the job queue. Jobs are dequeued from the backlog whenever a job is successfully allocated, migrated to the waiting queue or re-injected into the backlog. Finally, a job can arrive into the backlog at each time step.
- The job queue holds the jobs ready for allocation. It is an array with a predefined size which represents the amount of jobs it can hold. For example, if the job queue has a size of five, five different jobs can be placed in the queue. Once a job is in the queue it can either be allocated to the edge instance, be migrated to the cloud instances or placed in the waiting queue.
- The waiting queue has a similar structure as the job queue but differs in terms of purpose and actions. The waiting queue offers a way for the agent to temporarily set aside any jobs that could hinder the slowdown of the workload. A job in the waiting queue cannot be allocated, instead it can either be re-injected into the backlog or in one of the slots in the job queue.
- Machine 1 and machine 2 are lists that hold the allocated jobs for a length of time equal to j_L attribute of the workload allocated. Machine 1 represent the edge instance while machine 2 depicts a cluster of cloud instances.

The actions available: The original version of DeepRM contained two different actions, "MoveOn" and "Allocate", in order to deal with anomalous data we added three more actions for the agent to interact with the environment as seen in diagram4.

- MoveOn: Move on to the next timestep: $t = t + 1$.
- Allocate: Allocate job to machine 1.
- Migrate: Allocate job to machine 2.
- Remove: Send jobs to the waiting queue.
- Re-inject: Send jobs from the waiting queue to the backlog.

Allocation system: DeepRM provides an allocation system where jobs can only be allocated if there are sufficient resources r at the time t and time horizon of allocation. Figure3 illustrates the allocation process. In figure 3 there are three distinct jobs waiting to be allocated. Each job requires an amount of different types of resource, either CPU or Memory. Jobs also have a length j_L which represents the time that they will take to finish j_F. For example, the job in slot 1, requires two units of CPU, one unit of Memory and takes 3 units of time to finish, if any of these requirements are not met then it cannot be allocated. The machine can hold any number of jobs as long as the resource requirements are met and the time horizon constraint is respected. For example, in figure3 the job in slot 1 cannot be allocated as it requires 3 units of time and 2 units of CPU which exceeds the current time horizon and resource available in the machine. On the flip side, the job in slot 3 could be allocated at timestep $t + 3$ as it meets the resource and length constraints. In DeepRM_AD we adopt the same allocation system as DeepRM.

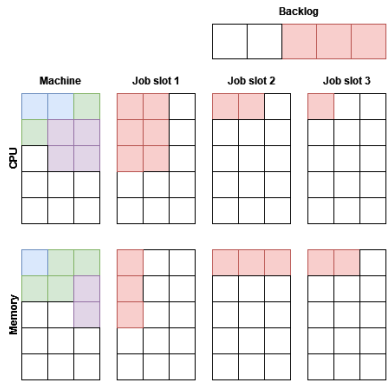


Figure 3: Allocation mechanism in DeepRM

Reward system: DeepRM is an algorithm focused primarily around optimal job scheduling and slowdown. As such, the reward system in DeepRM is straightforward and consists of penalizing the agent in each state to force the agent to finish the scheduling as soon as possible to avoid further penalties. While this reward system performs well to optimize the slowdown of jobs, it remains very linear and would perform badly for multi-objective problems. As we are aiming to detect anomalies and optimize the slowdown of jobs, our task now becomes maximizing the rate of anomalies detected while minimizing the slowdown of jobs. So instead we offer a reward system that balances negative and positive rewards such that the agent is rewarded for detecting anomalies and penalized on the slowdown and for migrating non-anomalous workload to the cloud. The reward system of DeepRM_AD is described in pseudo-code10 and in the following formula:

$$R = \sum_{j \in J} \frac{-1}{j_L} + \sum_{j \in M} \Big[I\left(\max(\vec{r_j}) \geq A_{res}\right) \frac{1}{\max(\vec{r_j})} + I\left(L_j \geq A_{len}\right) \frac{1}{j_L} + I\left(\max(\vec{r_j}) < A_{res} \wedge j_L < A_{len}\right) - 1 \times 5 \Big]$$

where R represents the total reward computed at each time step, $\sum_{j \in J} \frac{-1}{L_j}$ represents all the jobs in the backlog, waiting queue, job slot and remove queue where for each jobs we compute -1 divided

by the jobs length j_L and sum these quantities. In the second part of the reward function we compute the sum of all jobs j that have been migrated in M where if either their maximum resource demanded $\max(\vec{r_j})$ or their job length j_L is equal or greater than the anomalous thresholds set for resource requested A_{res} or job length A_{len}, we then compute $\frac{1}{\max(\vec{r_j})}$ and $\frac{1}{L_j}$ respectively. On the flip-side if a job is incorrectly recognized as anomalous and migrated in M we penalize the agent by giving a penalty of -5. As such, we incentivize the agent to correctly detect anomalous workload while minimizing the slowdown.

State observation: The state is represented by values in the range of 0 to 1 in a 3D array. The first dimension of the array models the observation of the second and third dimension over the time horizon. The second and third dimensions represent the resource requirement and utilization of $r1$ and $r2$ respectively and is modelled based on the size of the job queue, machine 1 and backlog. In the state observation, jobs waiting to be ran are represented by 1s, while empty spaces by 0s and running jobs are unique values between 0 and 1. To optimize the training of our model we remodeled the second and third dimension of the state observation to take into account the added components. As such, waiting queue and machine 2 components were added to the state representation of the environment which doubled the size of the state.

Imitation learning: Imitation learning is used by DeepRM to provide a baseline policy that will later be refined by the RL agent. Using imitation learning can drastically improve the training and convergence towards an optimal policy as a good enough policy is already provided by a heuristic such as SJF[1]. Despite heuristics being provided in the source code of DeepRM, we had to modify them to take into account the added actions, components and constraints in DeepRM_AD. Consequently, we offer an adapted version of SJF that can grasps the modifications made in our environment as seen in pseudo-code9.

4.2.2 *DeepRM ECO:.*

DeepRM_ECO stands for edge cloud optimization. The implementation simulates a simplified edge-cloud continuum dynamic in which jobs can either be allocated to two edge instances with limited resource or a cloud instance with a higher resource availability. Hence, with DeepRM_ECO we aim to test anomaly detection in an environment where strict constraints are in place. This approach takes advantage of the scale out nature of distributed systems to optimize the job scheduling process and forces strict resource constraints upon each instances to mimic the diverse computing power in the continuum.

Architecture: The architecture of DeepRM_ECO is comparable to DeepRM in the sense that it utilizes the same components but differ due to the scale out nature of DeepRM_ECO as seen in figure5.

DeepRM_ECO components:

- The backlog
- The Job queue component is similar to DeepRM_AD but does not have access to the "remove" action. Instead, three different allocation actions are available to allocate jobs in any of the three machines.
- Three different machines with various resource capacities. For example, machine 1 and 2 could be thought of as an edge instances with limited computing resources like 10 units of CPU and memory. Machine 3 would represent a more powerful hardware from a data center with 15 units of CPU and memory.

Actions available: The same basic actions as DeepRM are available to the agent. But instead of having only one allocation action it now has three corresponding to the three different machines.

Reward system: The reward system is more straightforward than DeepRM_AD due to the resource constraints placed on the machines. With DeepRM_ECO a less sensible balance of positive and negative rewards is given to the agent. Indeed the resource limit placed on each machine takes care of migrating anomalous jobs to the right machine. As such, the problem becomes more of a minimization one, but positive rewards are still given to help guide the agent in detecting anomalous tasks based on length. The reward system can be viewed in pseudo-code11 and in the following formula:

$$R = \sum_{j \in J} \frac{-1}{j_L} + \sum_{j \in M} \left[I\left(j_L \geq A_{len}\right) 1 \times 3 + I\left(j_L < A_{len}\right) 1 \times -2 \right]$$

The formula is also calculated at each time step and is similar but simpler than DeepRM_AD's due to the constraints set. J represents all jobs in the backlog, the job slot, machine 1 and machine 2. M represents machine 3, the cloud instance where the anomalous workload should be migrated. If the job migrated has a length j_L equal or above the anomaly threshold A_len we give the agent a reward of 3, if the job is incorrectly identified as anomalous we give it a penalty of -2. *State observation:* The state observation is similar but smaller than DeepRM_AD as only the representation of the new machines had to be added.

Imitation learning: To find the baseline policy of DeepRM_ECO we implemented a second variation of the SJF heuristic in order to correctly interact with our new environment. The modified SJF heuristic is described in pseudo-code 12.

4.2.3 *The Policy Gradient Network and objective function.*

The output layer of the neural network was modified for each implementations to conform to the number of actions available for both implementations. In order to maximize our objective function: $E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$3 we utilize the policy gradient method and the gradient descent function used by DeepRM[9] described in 3.

4.3 Evaluation method:

The evaluation will compare DeepRM_ECO, DeepRM_AD, DeepRM_AD without the remove queue component and DeepRM in order to quantify the performance of each models relative to the others.

4.3.1 *Job slowdown:*

To evaluate the job slowdown, we will compare the average slowdown of the different implementations in different stress test scenarios. The average slowdown will be calculated as such: $S_J = J_F / J_L$ where J_F is the completion time of all jobs summed, and J_L is sum of the duration of all completed jobs. The scenarios will be distinct from one another in regard to their simulation length, anomalous job creation rate and mode. Each of the implementations will be tested on three different sequence of randomly generated jobs, where for each sequence we run 20 monte-carlo simulations on 15 different simulation length going from 50 to 190 with an increment of 10 between simulation length.

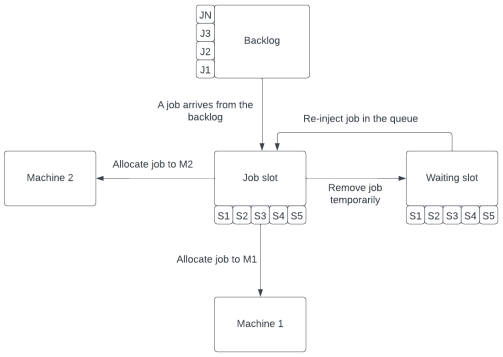


Figure 4: Architecture of DeepRM_AD

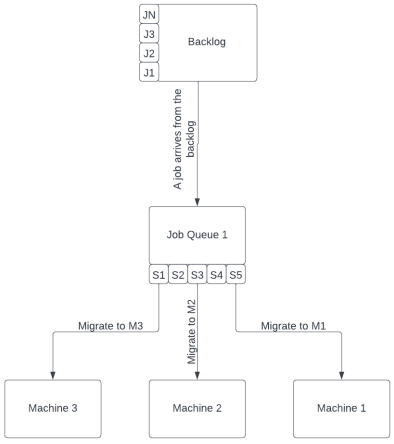


Figure 5: Architecture of DeepRM_ECO

These experiments will be replicated three times to test the performance of the models when the anomaly creation rate is 0.1, 0.3 and 0.5. Finally, all these experiments will be running on two different modes of DeepRM: "all_done" and "no_job_left" where the mode "all_done" waits for every job to be finished before terminating the process while the mode "no_jobs_left" terminates when the current time step is equal to the simulation length.

4.3.2 Anomaly detection: To assess the ability of our models to detect and treat anomalies, we will score the predictions made by our model with respect to their ground truth. The information collected from the slowdown test will serve to compare the amount of anomalies detected in when the anomaly rate is at 0.1, 0.3 and 0.5. Finally, we are going to use precision, recall and F1-Score as the metrics to evaluate the anomaly detection module with an emphasis placed on the recall metric.

4.3.3 Experimental setup: The experiments will be ran on two separate instances created on the Openstack environment of EscherCloudAI to optimize the time taken to run the tests. The hardware provided by EscherCloudAI can be seen in table4. DeepRM_ECO ran on the first instance while DeepRM_AD on the second one.

To collect and analyse the generated data we added scripts to write the data into csv files.

As many different experiments with various parameters will be ran, we chose to use DagsHub for its experiment versioning and tracking capabilities. A script was also built into all versions of DeepRM to collect the metrics and parameters of our experiments such that experiments can be correctly added to the DagsHub repository.

Every experiments are reproducible thanks to the parameters recorded in the DagsHub repository. A seed is also present in the code base of each implementations to ensure that the sequence of random jobs generated for the training are the same for every implementations.

5 RESULTS

5.1 Job Slowdown.

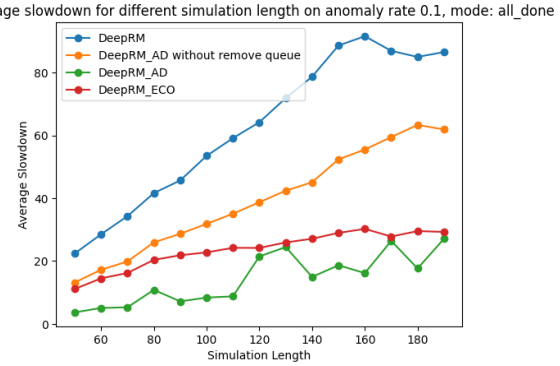
We first compare the job slowdown between the different implementations and the authentic version of DeepRM[9].

5.1.1 Training results. To quantify the performance of the models we look at their mean slowdown over a 3000 epochs training session. We report these results in figure6. Overall, every implementation outperforms the baseline DeepRM. On the 3000th epochs DeepRM_AD reached a mean slowdown S_J4 of 3.3, DeepRM_AD without the remove queue reached 5.2 and DeepRM_ECO reached a mean slowdown of 4.4. We find with these results that the variants of DeepRM perform significantly better than the original DeepRM who reached a mean slowdown of 9.7 on the last epoch.

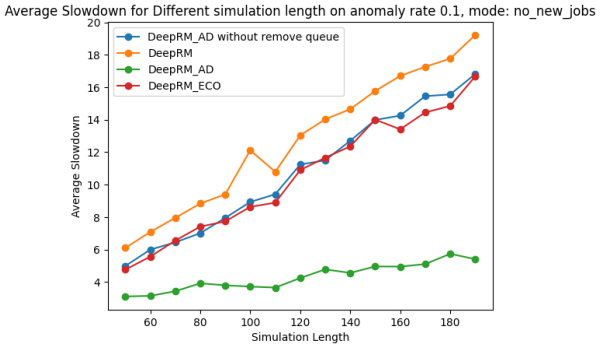
Figure 6: Mean slowdown during training for DeepRM (blue), DeepRM_AD (green), DeemRM_AD without the remove queue (orange) and DeepRM_ECO (red)

5.1.2 Test results. We performed extensive testing on different seeds, simulation length, mode and anomaly arrival rate to ensure that the models can adapt to different scenarios. The results of these tests are summarized in figure7a, 7b and 13. In general the original DeepRM is outclassed regardless of the anomaly rate, simulation

length or mode. Interestingly DeepRM_AD outperformed every implementation by a margin when using the mode "no_jobs_left" as seen in figure7b and 13. In contrary figure7a and 13 demonstrate that DeepRM_ECO and DeepRM_AD without the remove queue performed significantly better on the "all_done" mode.



(a) Mean job slowdown on varying simulation length at anomaly rate 0.1 on mode all_done



(b) Mean job slowdown on varying simulation length at anomaly rate 0.1 on mode no_new_job

5.2 Anomaly Detection.

5.2.1 Analysis of the anomaly detection module. To objectively evaluate the performance of the models for detecting and migrating anomalies using widely adopted evaluation metrics including recall, precision and F1 Score. The evaluation was conducted on more than 140 000 data points for each model and for every anomaly rate investigated.

In table1, we compare the performance of our models on an anomaly rate of 0.1. All three models show a low precision but high recall scores showing that they often misclassify normal tasks as anomalous and yet are proficient at correctly identifying anomalies.

When the anomaly rate is set to 0.3, all models improve drastically in precision, recall and F1 score as seen in Table2. At this anomaly rate, DeepRM_ECO once again exhibits almost a perfect recall and a much higher precision and F1 Score than previously.

Finally, at the highest anomaly rate of 0.5, the models once again improves in almost every metrics while maintaining a high recall as presented in Table3. The two variation of DeepRM_AD largely outperformed in precision while scoring satisfactory recall and F1 Score.

Table 1: Anomaly Detection Model Evaluation

Anomaly rate 0.1	Scores		
	Precision	Recall	F1 Score
DeepRM_AD	0.361	0.793	0.496
DeepRM_AD no remove queue	0.304	0.748	0.432
DeepRM_ECO	0.305	0.998	0.468

Table 2: Anomaly Detection Model Evaluation

Anomaly rate 0.3	Scores		
	Precision	Recall	F1 Score
DeepRM_AD	0.796	0.796	0.796
DeepRM_AD no remove queue	0.735	0.824	0.777
DeepRM_ECO	0.733	0.985	0.840

Table 3: Anomaly Detection Model Evaluation

Anomaly rate 0.5	Scores		
	Precision	Recall	F1 Score
DeepRM_AD	0.943	0.791	0.860
DeepRM_AD no remove queue	0.919	0.897	0.908
DeepRM_ECO	0.842	0.952	0.893

With these experiments we demonstrated that the models have different strengths. The DeepRM_AD models showed to excel in precision especially as the anomaly rate increases, while DeepRM_ECO showcased a robust recall over the three experiments while keeping a satisfactory precision and F1 Score in experiments with high anomaly rates.

6 DISCUSSION

6.1 Reflection

In this study we have proposed two models capable of consistently treating anomalous data across different scenarios and modes, all of it while maintaining a lower slowdown than the baseline DeepRM.

The recall obtained across the experiments are promising and highlight the potential of DRL as an alternative to other anomaly detection and classification algorithms. Indeed our implementations scored relatively high in recall indicating that anomalies were in most cases correctly detected and treated. Our results also show that while DRL performed better when strict constraints were in place, the agents could also reach tolerable results when the environment is bounded to limited constraints. With this insight we confirmed the adaptability of DRL agents which further support the potential use of DRL algorithms to treat anomalous tasks and the job slowdown in the cloud-edge continuum.

With the experiments we also highlight a low precision and F1 Score in low anomaly rate scenarios. While this result could indicate a failure in some cases, in the context of distributing jobs across the system such metric matters less than recall as our goal is to correctly detect anomalies and minimize the slowdown. For

example, in a realistic case, if the capacity of an edge node is full, jobs will be sent to a different node either edge or cloud if it means minimizing the slowdown. Therefore non anomalous jobs can be migrated if it leads to a more optimal scheduling policy.

Overall, in this project we have demonstrated a new way to optimize the cloud-edge continuum through a learning-based anomaly-tolerant resource management method that leverages the power of DRL to simultaneously manage resources and detect anomalies while using the state observation of the environment as input to the algorithm to optimize the scheduling and anomaly handling.

6.2 Limitations

While we have explored and proposed a solution for concurrently scheduling tasks and detecting anomalies, we have done so in a limited test setting with synthetic data. Such limits prevents us from confirming the performance of a DRL algorithm in optimizing the cloud-edge continuum as our test environment and synthetic data is heavily abstracted compared to a real world setting. This limitation is important to consider as an increase in the complexity of the environment and data could lead to higher computational complexity due to the increased size of the state, longer training time, a more complex reward function and difficulties in integrating new components in the environment.

Nevertheless, some of these limitations could be dealt with. For example the use of a convolutional neural network (CNN) could potentially better capture the state observation which could lead to faster training time and convergence towards an optimal reward. The increasing complexity could be tackled in many ways such as load balancing the training of the DRL model on a cluster of multiple instances.

Another limit encountered during the project was the efficiency and legitimacy of our testing methods for both detecting anomalies and minimizing the slowdown. The DeepRM model we built upon did not contain a component to treat anomalies, for this reason we implemented three variations of DeepRM capable of dealing with fraudulent data but still lacked a solid baseline to compare with. While, this approach still allowed us to demonstrate the treatment of anomalies in different settings, it would have been more insightful to compare it with some state of the art models. In addition, as our implementations used additional instances to treat anomalies. As such, the job slowdown of our models was more than likely going to be lower than baseline DeepRM who uses a single instance to schedule jobs.

Finding the optimal reward function also proved to be a great limit to our work as finding the right balance of reward for our agent to minimize the slowdown while maximizing the anomalies treated often led to unpredictable, suboptimal and contradictory behaviours due to the difference in nature of both problems.

7 CONCLUSION

This paper proposed and explored the use of DRL as a learning-based anomaly-tolerant resource management method through two DRL implementations based on DeepRM. We have analyzed the performance and adaptability of our algorithms in various scenarios. In the end our findings underscored the potential of DRL in optimizing the cloud-edge continuum by using the state observation of the environment as input to dynamically and simultaneously regulate scheduling and detecting anomalies.

While promising results were reached, the project was still subject to a number of limitations. The two primary ones being complexity of implementing and training a multi-objective model that harnesses the benefits of DRL and the simplification of the environment and data. To tackle these limitations future work could involve the use of a CNN to better capture the state of the environment hence potentially reducing the training time. A small but realistic cloud-edge test environment composed of virtual machines could also be built to mimic a more world-like scenario in which synthetic but more realistic tasks could be scheduled.

This research will hopefully contribute to our evolving understanding of the cloud-edge continuum optimization.

REFERENCES

[1] Muhammad Akhtar, Bushra Hamid, Inayat Ur-Rehman, Mamoona Humayun, Maryam Hamayun, and Hira Khurshid. 2015. An Optimized Shortest job first Scheduling Algorithm for CPU Scheduling. *J. Appl. Environ. Biol. Sci. , 5(12)42-46, 2015* 5 (01 2015), 42–46.

[2] S Chithra, D Maheswari, and Chithradevi Sethurathinam. 2022. A Comparative Study on Cloud Computing and Edge Computing with its Applications. 12 (02 2022).

[3] Gabriel Dulac-Arnold, Daniel J. Mankowitz, and Todd Hester. 2019. Challenges of Real-World Reinforcement Learning. *CoRR* abs/1904.12901 (2019). arXiv:1904.12901 http://arxiv.org/abs/1904.12901

[4] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, Joelle Pineau, et al. 2018. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning* 11, 3-4 (2018), 219–354.

[5] Richard Graham, Sung-Eun Choi, David Daniel, Nehal Desai, Ronald Minnich, Craig Rasmussen, L. Risinger, and Mitchel Sukalski. 2002. A network-failure-tolerant message-passing system for terascale clusters. 77. https://doi.org/10.1145/514203.514205

[6] Rajkumar Buyya Guangyao Zhou, Wenhong Tian. 2021. Deep Reinforcement Learning-based Methods for Resource Scheduling in Cloud Computing: A Review and Future Directions. *ACM Computing Surveys, 54(4), Article 76.* (2021).

[7] Wenxia Guo, Wenhong Tian, Yufei Ye, Lingxiao Xu, and Kui Wu. 2021. Cloud Resource Scheduling With Deep Reinforcement Learning and Imitation Learning. *IEEE Internet of Things Journal* (2021).

[8] Songqiao Han, Xiyang Hu, Hailiang Huang, Mingqi Jiang, and Yue Zhao. 2022. ADBench: Anomaly Detection Benchmark. https://doi.org/10.48550/ARXIV.2206.09426

[9] Ishai Menache Srikanth Kandula Hongzi Mao, Mohammad Alizadeh. 2016. DeepRM: A Deep Reinforcement Learning Framework for Intelligent Resource Management in Cloud Computing. *In Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets) (pp. 50-56)* (2016).

[10] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.

[11] Sandeep Kelkar. 2015. Challenges and Opportunities with Cloud Computing. *International Journal of Innovative Research in Computer and Communication Engineering* 3 (04 2015), 2719–2724. https://doi.org/10.15680/ijircce.2015.0304007

[12] Ashish Kumar. 2015. Cloud Computing Challenges: A Survey. *International Journal of Computer Science Engineering Technology* 6 (10 2015), 602.

[13] Abdeldjalil Ledmi, Hakim Bendjenna, and Sofiane Mounine Hemam. 2018. Fault Tolerance in Distributed Systems: A Survey. In *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*. 1–5. https://doi.org/10.1109/PAIS.2018.8598484

[14] Jia-Chun Lin and Ming-Chang Lee. 2016. Performance Evaluation of Job Schedulers on Hadoop YARN. 28, 9 (jun 2016), 2711–2728. https://doi.org/10.1002/cpe.3736

[15] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (jan 1973), 46–61. https://doi.org/10.1145/321738.321743

[16] Ning Liu, Zhe Li, Zhiyuan Xu, Jielong Xu, Sheng Lin, Qinru Qiu, Jian Tang, and Yanzhi Wang. 2017. A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning. *CoRR* abs/1703.04221 (2017). arXiv:1703.04221 http://arxiv.org/abs/1703.04221

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015.

Human-level control through deep reinforcement learning. *Nature* 518 (02 2015), 529–33. https://doi.org/10.1038/nature14236

[18] Brendan O'Donoghue, Ian Osband, Rémi Munos, and Volodymyr Mnih. 2017. The Uncertainty Bellman Equation and Exploration. *CoRR* abs/1709.05380 (2017). arXiv:1709.05380 http://arxiv.org/abs/1709.05380

[19] Frans A. Oliehoek, Matthijs T. J. Spaan, and Nikos Vlassis. 2011. Optimal and Approximate Q-value Functions for Decentralized POMDPs. *CoRR* abs/1111.0062 (2011). arXiv:1111.0062 http://arxiv.org/abs/1111.0062

[20] Suraj Pandey, Linlin Wu, Siddeswara Guru, and Rajkumar Buyya. 2010. A Particle Swarm Optimization-Based Heuristic for Scheduling Workflow Applications in Cloud Computing Environments. *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, 400–407. https://doi.org/10.1109/AINA.2010.31

[21] Rasmus V. Rasmussen and Michael Trick. 1999. Round Robin Scheduling - A Survey. (12 1999). https://doi.org/10.1184/R1/6707867.v1

[22] SchedMD. 2019. Slurm Workload Manager - Fair Tree Fairshare Algorithm. https://slurm.schedmd.com/fair_tree.html. Accessed: 2023-06-20.

[23] Uwe Schwiegelshohn and Ramin Yahyapour. 1998. Analysis of First-Come-First-Serve Parallel Job Scheduling. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms* (06 1998). https://doi.org/10.1145/314613.315031

[24] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[25] Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. 2021. PreGAN: Preemptive Migration Prediction Network for Proactive Fault-Tolerant Edge Computing. https://doi.org/10.48550/ARXIV.2112.02292

[26] J.D. Ullman. 1975. NP-complete scheduling problems. *J. Comput. System Sci.* 10, 3 (1975), 384–393. https://doi.org/10.1016/S0022-0000(75)80008-0

[27] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France.

[28] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*.

[29] J Xu, Z Xu, and B Shi. 2022. Deep Reinforcement Learning Based Resource Allocation Strategy in Cloud-Edge Computing System. *Frontiers in Bioengineering and Biotechnology* 10 (2022), 908056. https://doi.org/10.3389/fbioe.2022.908056

[30] Chuan Wu Yixin Bao, anghua Peng. 2019. Deep Learning-based Job Placement in Distributed Machine Learning Clusters. (2019).

[31] Yingchen Yu. 2021. A Research Review on Job Shop Scheduling Problem. *E3S Web of Conferences* 253 (01 2021), 02024. https://doi.org/10.1051/e3sconf/202125302024

Appendix A PSEUDO-CODES

```
Set num_res to 2
Set anomalous_job_rate to 0.1
Set job_small_chance to 0.50
Set job_len to 18
Set job_len_big_lower to job_len * 2 / 3
Set job_len_big_upper to job_len
Set job_len_small_lower to 1
Set job_len_small_upper to job_len / 5
Set max_nw_size to 10
Set dominant_res_lower to max_nw_size / 2
Set dominant_res_upper to max_nw_size
Set other_res_lower to 1
Set other_res_upper to max_nw_size / 2
Set anomalous_job_len_upper to 29
Set anomalous_job_len_middle to 24
Set anomalous_job_len_lower to 19
Set anomalous_resource_lower to 11
Set anomalous_resource_upper to 15

Function job_generation:

    Set random_val to a random number between 0 and 1

    // Determining job length
    If random_val is less than job_small_chance AND random_val is greater than anomalous_job_rate: // for small job
        Set nw_len to a random integer between job_len_small_lower and job_len_small_upper inclusive

    Else If random_val is less than anomalous_job_rate: // for anomalous job
        Set nw_len to a random integer between anomalous_job_len_lower and anomalous_job_len_upper

    Else: // for big job
        Set nw_len to a random integer between job_len_big_lower and job_len_big_upper inclusive

    Initialize nw_size as an array of zeros with length num_res

    // Determining job resource request
    Set dominant_res to a random integer between 0 and num_res
    Set random_res to a random number between 0 and 1

    For each i in the range from 0 to num_res:
        If i is equal to dominant_res:
            If random_res is greater than anomalous_job_rate:
                Set nw_size[i] to a random integer between dominant_res_lower and dominant_res_upper inclusive
            Else:
                Set nw_size[i] to a random integer between anomalous_resource_lower and anomalous_resource_upper

        Else:
            Set nw_size[i] to a random integer between other_res_lower and other_res_upper inclusive

    Return nw_len and nw_size
```

Figure 8: pseudo-code for data generation in DeepRM_AD and ECO

Appendix B EXPERIMENTAL SETUP

	VCPU	GPU	GPU Memory (GB)	RAM (GB)	Disk Space (GB)
Instance 1	4	Nvidia A100	10	32	20
Instance 2	12	Nvidia A100	40	96	20

Table 4: Hardware Configuration for Instances

Appendix C SLOWDOWN FIGURES

```
Initialize anomaly_length_lower_bound, anomaly_length_upper_bound, anomaly_length_middle_bound, anomaly_resource_lower_bound, job_slot, removed_slot, avbl_res

Algorithm modified_get_sjf_action
  Input: machine, job_slot, removed_slots, pa
  Output: action to be performed (act)

  Initialize sjf_score to 0
  Set default act as the number of slots in job_slot (If no action available, move on)

  For each slot i in job_slot.slot:
    Set new_job to job_slot.slot[i]
    If new_job exists (there is a pending job):
      Calculate available resources (avbl_res) for the new_job's length
      Calculate remaining resources (res_left) after allocating for job

      If res_left >= and job ressources < anomaly_resource_lower_bound:

        If new_job length is < anomaly_length_lower_bound or new_job is in machine's re_allocated_jobs list:
          Calculate temporary SJF score (tmp_sjf_score)

          If tmp_sjf_score > sjf_score:
            Set sjf_score to tmp_sjf_score
            Set act to i
            Return act (allocate)

          Else if new_job is in re_allocated_jobs and tmp_sjf_score > sjf_score:
            Set sjf_score to tmp_sjf_score, remove job from re_allocated_jobs
            Set act to i
            Return act (allocate)

        Else if new_job length is >= to anomaly_length_middle_bound:
          Set act to len(job_slot.slot) * 2 + i + 1
          Return act (migration)

        Else if new_job length is in between anomaly_length_lower_bound and anomaly_length_middle_bound and new_job is not in re_allocated_jobs:
          Add job to re_allocated_jobs
          Set act to len(job_slot.slot) + i + 1 (removal actions)
          Return act (remove)

      Else if new_job resources are >= to anomaly_resource_lower_bound:
        Set act to len(job_slot.slot) * 2 + i + 1 (migration actions)
        Return act (migration)

  Else:
    If no new_job in job_slot and there are jobs in removed_slots:
      For each slots j in removed_slots.slot[j]:
        If slot[j] is not None:
          Set act to len(job_slot.slot) * 3 + j + 1 (injection actions)
          Return act (re-inject)

    If act remains default (no specific action determined):
      Return act (Move on)

  If no suitable action is found within the loop, return act (move on which will hold its initial value)
```

Figure 9: pseudo-code of modified version of SJF for DeepRM_AD

```
Initialize: delay penalty, hold penalty, dismiss penalty, removed penalty, migrate penalty as -1

Define function get_reward:

    Initialize reward as 0

    For each job 'j' in running jobs of the machine:
        Add delay penalty divided by job length to reward

    For each slot 'j' in job slot:
        If slot 'j' is not None:
            Add hold penalty divided by job length to reward

    For each job 'j' in job backlog:
        If job 'j' is not None:
            Add dismiss penalty divided by job length to reward

    For each slot 'j' in remove slot:
        If slot 'j' is not None:
            Add removed penalty divided by job length to reward

    For each job 'j' in running_jobs2 jobs of the machine2:
        If max resources of job 'j' is greater or equal to anomalous job resources lower limit:
            Add migrate penalty divided by negative of max resources of job 'j' to reward
        Else If job length is greater or equal to anomalous job length middle limit:
            Add migrate penalty divided by negative of job length to reward
        Else:
            Add migrate penalty times 6 to reward

    Return reward
```

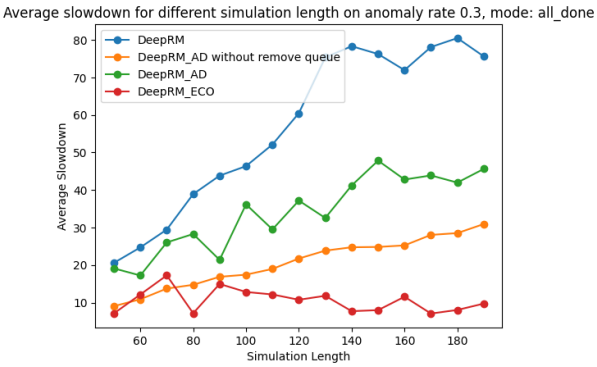
Figure 10: modified version of reward system for DeepRM_AD

```
function get_reward
    initialize reward to 0
    initialize delay_penalty to -1
    initialize hold_penalty to -1
    initialize dismiss_penalty to -1

    for each job j in machine.running_job
        increment reward by delay_penalty divided by job length

    for each job j in machine.running_job2
        increment reward by delay_penalty divided by job length

    for each job j in machine.running_job3
        if job length is greater or equal to anomalous_job_len_lower_bound
            increment reward by delay_penalty multiplied by -3
        else
            increment reward by delay_penalty multiplied by 2

    for each job j in job_slot1.slot
        if job j is not None
            increment reward by hold_penalty divided by job length

    for each job j in job_backlog.backlog
        if job j is not None
            increment reward by dismiss_penalty divided by job length

    return reward
```

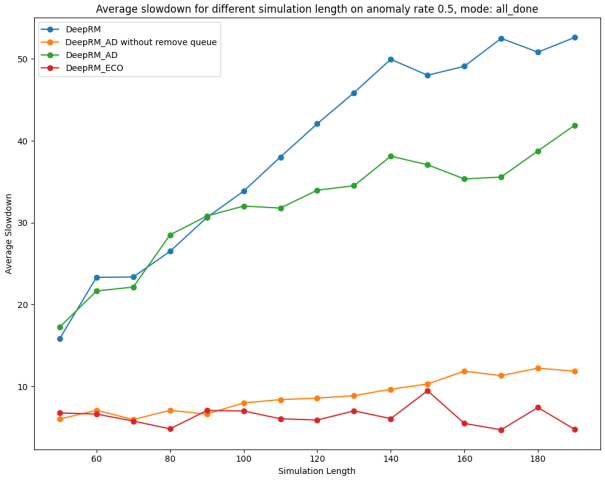
Figure 11: modified version of reward system for DeepRM_ECO

```
Initialize anomaly_length_lower_bound, anomaly_length_upper_bound, anomaly_length_middle_bound, anomaly_resource_lower_bound, job_slot, avbl_res1, avbl_res2, avbl_res3

Algorithm get_sjf_action
  Input: machine, job_slot:
  Output: action to be performed (act)

    Initialize sjf_score1, sjf_score2, sjf_score3 to 0
    Set default act as the number of slots in job_slot (If no action available, hold)

    For each slot i in job_slot.slot:
        Set new_job to job_slot.slot[i]
        If new_job exists:
            Calculate available resources for each machine slot (avbl_res1, avbl_res2, avbl_res3) for the new_job's length
            Calculate remaining resources for each machine slot after potential allocation of new_job (res_left1, res_left2, res_left3)

            If res_left1 is all >= 0 and new_job's length is >= anomalous_job_len_lower_bound and new_job's resources are < anomalous_job_resources_lower_bound:

                Calculate the Shortest Job First (SJF) score for the job as 1 / new_job.length (tmp_sjf_score1)

                If tmp_sjf_score1 > sjf_score1:

                    Set sjf_score1 to tmp_sjf_score1
                    Set act to i
                    Return act (allocate in machine 1)

            Else if res_left2 is all >= 0 and new_job's length is < anomalous_job_len_lower_bound and new_job's resources are < anomalous_job_resources_lower_bound:

                Calculate the Shortest Job First (SJF) score for the job as 1 / new_job.length (tmp_sjf_score2)

                If tmp_sjf_score2 > sjf_score2:

                    Set sjf_score2 to tmp_sjf_score2
                    Set act to len(job_slot.slot) * 2 + i + 1
                    Return act (allocate in machine 2)

            Else if res_left3 is all >= 0 and any of new_job's resources are >= anomalous_job_resources_lower_bound:

                Calculate the Shortest Job First (SJF) score for the job as 1 / new_job.length (tmp_sjf_score3)

                If tmp_sjf_score3 > sjf_score3:

                    Set sjf_score3 to tmp_sjf_score3
                    Set act to len(job_slot.slot) * 3 + i + 1
                    Return act (allocate in machine 3)

    If no suitable action is found within the loop, return act (move on which will hold its initial value)
```

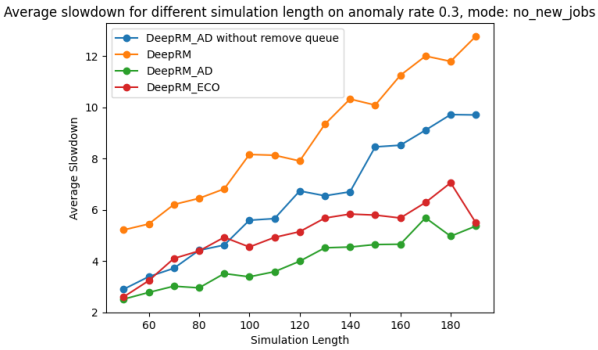
Figure 12: pseudo-code of modified version of SJF for DeepRM_ECO



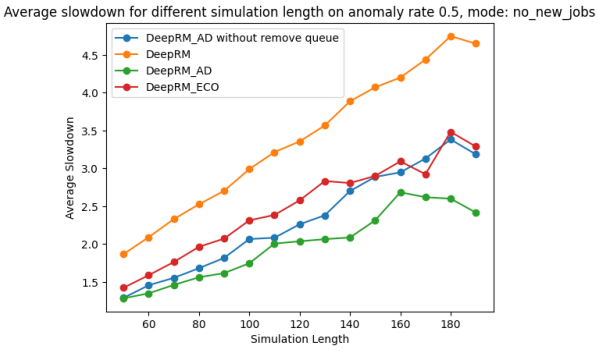
(a) Mean job slowdown on varying simulation length at anomaly rate 0.3 on mode all_done



(b) Mean job slowdown on varying simulation length at anomaly rate 0.5 on mode all_done



(c) Mean job slowdown on varying simulation length at anomaly rate 0.3 on mode no_new_jobs



(d) Mean job slowdown on varying simulation length at anomaly rate 0.5 on mode no_new_job

Figure 13: Results of the slowdown experiments