Received: 2 July 2025 | Revised: 17 December 2025 | Accepted: 6 January 2026 | Published online: 24 February 2026

Journal of Computational and Cognitive Engineering
2026, Vol. 00(00) 1–12
DOI: 10.47852/bonviewJCCE62026664

Research Article

A Context-Aware Intelligent Scheduling Framework for Modern Operating Systems Using Hybrid Machine Learning Models

Pandikumar Savarimalai[1,*], Menaka Chellappan[2], Murugaiyan Swaminathan Nidhya[3], Margaret Mary Thomas[4], Sevugapandi Nallathambi[5] and Manjula Selvaraj[6]

[1] *Department of Master of Computer Applications, Acharya Institute of Technology, India*

[2] *Department of Computer Applications, PES University, India*

[3] *School of Computer Applications, Dayananda Sagar University, India*

[4] *Department of Computer Science, Kristu Jayanti (Deemed to be University), India*

[5] *Department of Computer Science, Government Arts and Science College, India*

[6] *Department of Computer Science and Engineering, Vel Tech Rangarajan Dr. Sagunthala R&D Institute of Science and Technology, India*

Abstract: In modern computing environments, the operating systems are required to handle dynamic and complex applications with heterogeneous workloads. These workloads include interactive, data-centric, and high-computing applications. Interactive tasks require low latency and high response, while data-centric transactions demand high resource utilization and input/output. Likewise, high-computing processes including machine learning tasks require sustained throughput. Traditional CPU scheduling, like round-robin and priority scheduling, highly relies on static and rule-based systems. Generally, these give equal weightage to all processes. They are not paying attention to the contextual variations and working patterns that impact the responsiveness, efficiency, and overall performance. There is an evident need for a context-aware and adaptive CPU scheduling mechanism that will handle applications and processes based on context. To address this gap, this paper introduces an application-aware intelligent scheduling mechanism that integrates a random forest (RF) classifier and deep Q-network (DQN) reinforcement learning. The RF classifies the processes based on their working and behavioral patterns, and then DQN has system state and queue dynamics to make adaptive scheduling decisions. It can be continuously learned and adjusted based on the user's needs; it is responsive and achieves high throughput. Experimental evaluations are conducted on generative datasets and show remarkable performance improvements. The proposed framework achieves up to 42.2% responsive time reduction on interactive application workloads, 29.2% throughput gain against the machine learning training workloads, and 22.8% for database transactions. The architecture is computationally sparse and uses the existing Linux scheduling system. This renders it a convenient and scalable method of CPU scheduling in contemporary multicore systems.

Keywords: intelligence CPU scheduling, context-aware scheduling, reinforcement learning, random forest classification, deep Q-network

1. Introduction

Nowadays, managing workloads with diverse needs and performance levels is a challenge for contemporary operating systems [1]. Current computing environments handle applications that can range from ultra-fast real-time tasks to slower batch computing. In this context, regular scheduling algorithm designs are not adequate [2], as schedulers must handle cache affinity and different Non-Uniform Memory Access (NUMA) structures and assure that containers do not overlap [3].

The traditional methods used in CPU scheduling are not ideal for situations where web servers require fast responses, machine learning (ML) needs high throughput, and interactive applications require quick replies [4, 5]. Traditional CPU schedulers, including Completely Fair Scheduler (CFS), treat all applications the same, overlooking that each one has different needs. This causes resource allocation to be less efficient [6, 7]. These schedulers use general process measures and not the deeper

***Corresponding author:** Pandikumar Savarimalai, Department of Master of Computer Applications, Acharya Institute of Technology, India. Email: pandikumar2906@acharya.ac.in

traits of applications. Problems can arise where both slow and fast applications get the exact same attention [8].

These schedulers operate on fixed time quantum allocations and static priority mechanisms. They do not consider important workload characteristics such as CPU burst patterns, input/output (I/O) wait ratios, memory access patterns, or cache behavior. For instance, CFS uses a red–black tree structure to maintain process execution order based on virtual runtime (vruntime). It tries to give a just distribution of CPU time to all processes. Nevertheless, there is a significant drawback to this fairness-based approach. Interactive applications have short CPU bursts and I/O waits. They need scheduling right away when they are ready. Conversely, compute-intensive programs such as scientific simulations portray long CPU bursts with a small amount of I/O. They have the advantage of long continuous execution times so that they can use the cache to the maximum. CFS is not able to differentiate these two types of applications.

A round-robin scheduler gives each process a specified time slice in a circular queue. This duration of time is normally between 10 and 100 milliseconds. The long-running computational tasks are overly switched in terms of context by this approach. It also presents unwanted latency to interactive workloads that are sensitive to latency. Schedulers based on priorities provide a little bit of differentiation, with priority levels assigned on a manual basis. Nevertheless, they do not support dynamic adaptation to the changing behavior of the applications at run time and need a systematic administrator response.

Moreover, these conventional methods do not take into account NUMA topology consciousness. High-inter-thread communication processes in NUMA systems are to be assigned to cores that have the same memory node. This reduces access to remote memory. Cache affinity is also not taken into consideration in traditional schedulers. At times of rescheduling a process on the same core as it was running, the cache warm-up overhead gets to a minimum. These optimization opportunities are missed by the traditional schedulers.

There is a basic discrepancy between the potentials of the modern operating systems and the differentiation of behavioral needs of contemporary applications. The existing CPU scheduling strategies rest on an incorrect set of assumptions. They assume that one strategy of resource allocation can be helpful in all kinds of workloads. Nevertheless, the existence of facts proves otherwise. This non-context-dependent scheme brings about a considerable drop in performance with execution of the heterogeneous application in parallel [9, 10].

These scheduling problems could be addressed with intelligent systems, with the new advancements made in ML [11]. ML can detect the trends in how an application works, access previous outcomes, and change its scheduling choices based on the new reality instead of just operating under the fixed rules [12]. It happens that reinforcement learning proves to be particularly effective in the process of forming useful schedules, whereas random forest (RF) algorithms are effective when it comes to immediate data classification [13–15].

In this research, an Adaptive Context-Aware Reinforcement-Learning Scheduler (ACARS) framework is introduced, which addresses problems in handling heterogeneous workloads in modern operating system design by integrating RF classification with a deep Q-network (DQN). The system automatically sorts processes into five categories by analyzing their behavior patterns and then learns how to adjust the scheduling criteria as needed in a flexible multi-queue arrangement that operates alongside existing Linux processes. This framework significantly increases efficiency by providing faster response times for interactive programs and higher throughput for high computational jobs.

The rest of the paper is structured as follows: The literature study conducted in Section 2 investigates studies related to ML and CPU scheduling and hybrid ML approaches. Section 3 precisely points out the research's scope and defines questions. The principles of intelligent scheduler design and the mechanism of RF classification, covering state representation, Q-value calculation, and the way to combine the two models, are provided in Sections 4 and 5, respectively. The experimental setup and dataset description are presented in Section 6. Section 7 analyzes the outcomes of the experiment from various perspectives, and Section 8 is the conclusion of the paper, where the paper declares the obtained findings and future work.

2. Literature Review

2.1. Machine learning in CPU scheduling

ML integration in CPU scheduling has demonstrated substantial performance improvements across diverse computing environments [16]. Allaqband et al. [17] developed a comprehensive ML framework utilizing Long Short-Term Memory (LSTM), artificial neural network (ANN), and linear regression for heterogeneous multicore processors, achieving 1.2× performance enhancement and 20% throughput improvement through intelligent core-to-thread mapping. The results of the proposed ML-based scheduler were tested on selected heterogeneous multicore configurations only, which means that the outcome also strongly relies on the specific workloads and processor types applied and that only narrower generalizability is allowed. Gupta et al. [18] introduce an ANN-based scheduler that is able to improve the scheduling of CPU in heterogeneous systems, with the prediction of the behavior of applications that improves performance by 6.5–9.7% when compared to traditional approaches and, therefore, is effective at providing a task scheduling approach based on ML. Nevertheless, it is substantially based on simulated SPLASH-2 workloads and shares prediction error with the SQSP stage, meaning it is sensitive to non-regular application behavior and unreliable in a wide range of realistic conditions.

Deep-based reinforcement learning implementations have attained high gains. Han and Lee [19] implemented policy gradient reinforcement learning for Linux CPU scheduling on Android smartphones, demonstrating 3–11% energy savings while maintaining performance requirements. Their Learning-EAS framework dynamically adjusts CPU frequency and task migration policies based on real-time system analysis. The weakness of the study is that the Learning-EAS is purely based on on-device reinforcement learning. It is also conditioned to particular smartphones or workload patterns. Hence, its learned policies might not be easy to be generalized across varied real-life applications or unpredictable task behaviors. Nemirovsky et al.'s research [20] has shown that lightweight ANNs provide highly accurate performance predictions, yielding 25–31% throughput improvements over conventional heterogeneous schedulers. It is worth pointing out that the performance predictor and scheduler were tested on simulation and benchmark suites. They were not tested on actual deployment to a real system. Therefore, its accuracy and benefit could not be expected to be significant in the presence of real hardware variability and unpredictable workloads. Asad Hayat et al. [21] developed load-balanced task schedulers combined with ML-based device predictors that have reduced execution time by 65.63% and increased resource utilization by

93.3%. Nevertheless, only Polybench and AMD benchmark kernels are trained and tested on by the proposed scheduler. So it is unknown whether its predictive capability over devices and load-balancing capability will be effective for real-world workloads with unknown code patterns.

Recent optimizations focus on intelligent time quantum adjustment mechanisms. Shafi et al. [22] proposed adaptive quantum calculation methods responding to process characteristics and system load variations. In practice, Amended Dynamic Round Robin (ADRR) is validated only through numerical examples and MATLAB simulations on small, synthetic workloads. Sharma et al. [23] developed intelligent round-robin scheduling incorporating dynamic priority adjustments, demonstrating improved system responsiveness through quantum management and process prioritization. Multilevel Feedback Queue Evolution (MLFQ) systems have evolved toward intelligent implementations. Practically, it is only verified against one synthetic job pool whose burst times are fixed. Therefore, its clustering-based quantum selection cannot perform in a dynamic arrival or in the reality of workload fluctuation. Thombare et al. [24] presented dynamic time quantum allocation strategies demonstrating CPU utilization improvements through intelligent queue management. However, the procedure is only experimented on a small, simulated instance; thus, its dynamic-quantum advantages are not established on actual or diverse workloads. Higenson and Brown [25] proposed starvation mitigation methods that avoided starvation of the process but retained efficiency by using advanced priority management algorithms. The scheduling of priorities has optimized the use of smart assignment. It has made a contribution, yet the analysis is based solely on the results of the simulation of small, artificial workloads. Thus, the recommended mitigation could not be relied on to mitigate starvation and also to increase throughput during real-world and unforeseen process combinations. Kareem and Kumara [26] designed fuzzy logic schemes using a fuzzy inference system to calculate priority dynamically as opposed to the traditional non-dynamic methods in order to enhance responsiveness in the system. Its testing is however based on synthetic workloads only, and therefore, the effectiveness of it in the real world is not tested.

2.2. Hybrid machine learning approaches

Recent studies in hybrid ML designs have shown that multi-algorithm combinations have a lot of performance improvement. The hybrid scheduling that is proposed by Saha et al. [27], which combines red–black tree structure and Incremental Time Quantum Round Robin, has been proven to be the most efficient in terms of response time and resource usage. Nevertheless, it is not tested in the real world except using simulations, and therefore, its extension to a real-world scenario has not been proven. Patel and Solanki [28] reported that their Highest Response Ratio Next (HRRN) hybrid algorithm generates optimized results with zero job starvation. It is worth noting that the assessment is only based on a simulator that relies on a very simplistic one-CPU-burst process model and that they are not even testable in a real operating system, thus limiting the accuracy and practical effectiveness of the assessment. Zang et al. [29] developed a Hybrid Deep Neural Network Scheduler, achieving 9% better MAKESPAN performance than traditional approaches.

Muniswamy and Vignesh [30] proposed a hybrid container-cloud scheduler that integrates multi-swarm coyote optimization with deep learning predictors to dynamically allocate CPU resources, achieving improved response time and resource utilization over heuristic baselines. The scheduler is validated only through controlled container-cloud simulations and test vectors, with no real-world deployment to confirm robustness under unpredictable workloads. Cheng et al. [31] introduced a cost-aware real-time scheduler for hybrid cloud platforms using deep reinforcement learning (DRL), demonstrating superior Service Level Agreement (SLA) compliance and reduced operational cost through adaptive job placement and admission control. In practice, the method is evaluated solely in simulated hybrid cloud environments, leaving its real-system performance under heterogeneous latencies and dynamic arrivals unverified. Li et al. [32] presented GARLSched, a generative-adversarial DRL framework that synthesizes challenging workload scenarios during training, enabling the scheduler to maintain robustness and outperform standard DRL policies under diverse traces. However, it is trained and tested only on benchmark traces, so its effectiveness remains untested under real-time workload fluctuations and system-level uncertainties. Verma et al. [33] optimized Spark job scheduling using distributional deep learning, where DQN-based models capture latency variance more effectively, leading to improved throughput and higher deadline-meeting ratios in large-scale data-processing workflows. Despite its contributions, the evaluation is limited to controlled Spark workloads, without assessing behavior under large-scale cluster variability, node failures, or multi-tenant contention. Wang et al. [34] built on DRLIS, the DRL-based Internet of Things (IoT) application scheduler in fog-clouds, where response time and energy consumption are reported to be significantly reduced as compared to rule-based task allocation strategies. It relies on relatively fixed resources in the form of fog/edge and is only tested in simulated IoT and has no evidence to support its performance in the face of a more realistic device heterogeneity and network dynamics.

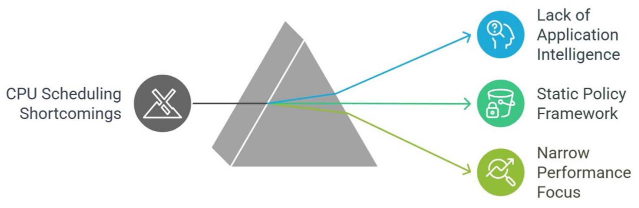
The existing literature has been severely limited in the space of practical implementations: the majority of the studies are based on simulation and are not validated on an actual basis, and application-based scheduling is not thoroughly examined, and in-depth models of ML integrations are underrepresented. Also, the lack of scalability analysis of the enterprise systems and the lack of standardized evaluation frameworks can make the performance comparisons not really meaningful.

3. Problem Analysis

3.1. Problem context

As the literature study states, the current Linux CPU scheduling solutions have a number of serious flaws that cause the inabilities of the systems to achieve the best performance, including a lack of application intelligence, a static policy framework, and a narrow performance focus (Figure 1).

Figure 1
Drawbacks of rule-based CPU scheduling



3.2. Research questions and contribution

1) How can CPU schedulers autonomously and accurately rank applications by their characteristics of execution and demand of resources?
2) How can CPU schedulers autonomously and accurately rank applications by their characteristics of execution and demand of resources?

The proposed research aims at developing and evaluating a new application-intelligent CPU scheduling framework that will benefit system performance by being smart and adaptive toward the intelligent management of the available resources and, at the same time, being compatible with the existing Linux scheduling architecture. The study will develop novel methods to use in identifying applications in scheduling environments, an adaptive policy modification system, and multi-goal optimization techniques for dynamic scheduling decisions.

4. Intelligent Scheduler Design

4.1. Multilevel queue architecture

The proposed scheduler employs a hierarchical four-tier queue system: Q_interactive for latency-critical applications (web servers, GUI applications), Q_standard for regular computational workloads (compilation, database operations), Q_background for CPU-intensive tasks (ML training, video encoding), and Q_system for system services and cleanup operations (Figure 2).

4.2. Process classification and assignment

When processes enter the runnable state, the scheduler extracts feature vectors encompassing CPU utilization, I/O behavior, memory usage, and system call signatures. The RF classifier maps processes to application categories using $f(p) = \operatorname{argmax} P(a|X(p))$, where $X(p)$ represents the feature vector and a denotes the predicted application type. Process assignment combines classification results with dynamic context information through priority scoring that considers temporal state, user activity, and system load conditions.

4.3. Adaptive scheduling operations

The scheduler picks which process can run next by utilizing a weighted service rate chosen based on queue position and the operating system's overall workload. For interactive applications, the CPU slices are shorter, so the computer responds better, but for computational tasks, slices are allocated for longer to prevent them from having to start over often. Performance tracking runs continuously in the system, and it updates parameters based on gradient-based methods in real time.

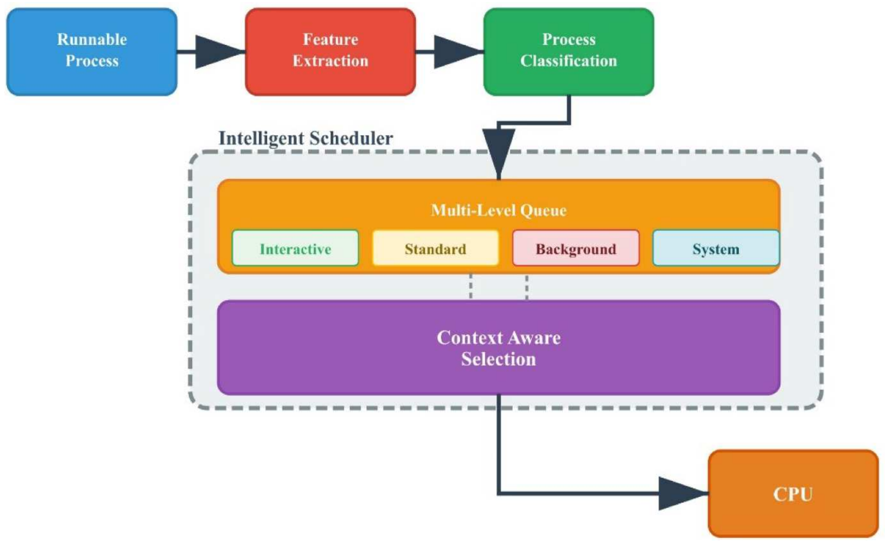
4.4. Dynamic queue management

Processes change queues when their priority ranking is higher than what the adaptive threshold allows, and hysteresis helps avoid frequent back-and-forth movement between queues. It takes into account how the CPU is used in the application and where and how its data is cached. With this system, scheduling is more accurate whenever classifications are right, and the machine continues to adjust and learn how to handle future categorization and scheduling tasks.

Algorithm 1: Main Application-Aware Scheduler

```
Input: System processes P, time interval $\Delta t$
Output: Optimized CPU scheduling decisions
1: Initialize: Q_high, Q_medium, Q_low, Q_idle $\leftarrow \emptyset$
2: Initialize: ClassificationModel M, ContextWeights W
3: while system_running do
4:     C(t) $\leftarrow$ UpdateSystemContext()
5:     P_new $\leftarrow$ GetNewRunnableProcesses()
6:     // Process Classification and Queue Assignment
7:     for each p $\in$ P_new do
8:         X(p) $\leftarrow$ ExtractFeatures(p)
9:         f(p) $\leftarrow$ M.classify(X(p))
```

Figure 2
System architecture



```
10:         $\pi$(p,t) ← CalculatePriority(f(p), C(t), W)
11:         Q_k ← AssignQueue($\pi$(p,t))
12:         Q_k.enqueue(p)
13:     end for
14:     // Scheduling Operations
15:     RebalanceQueues(C(t))
16:     Q_active ← SelectQueue(C(t))
17:     p_selected ← SelectProcess(Q_active, C(t))
18:     $\tau$(p_selected) ← CalculateTimeSlice(p_selected, C(t))
19:     Execute(p_selected, $\tau$(p_selected))
20:     CollectMetrics(p_selected)
21:     // Model Updates
22:     if t mod learning_interval = 0 then
23:         UpdateModels()
24:     end if
25:     t ← t + $\Delta$t
26: end while
```

5. Two-Stage ML Pipeline Design

5.1. Random forest process classification

When a new process enters the system, the classifier must quickly understand what kind of task it is dealing with. To achieve this, the RF model processes a set of 18 well-curated features that capture the behavior and resource profile of the process. These features include everything from CPU and memory usage to user interaction and system conditions:

F(p) = [process_name, command_arguments, thread_count, process_age, parent_process_name, user_id, cpu_usage_percent, memory_usage_mb, io_read_rate, io_write_rate, network_bytes_total, syscall_ frequency, time_of_day, system_cpu_load, system_memory_ free, user_input_recent, queue_wait_time, response_time]

The model runs these features through an ensemble of K decision trees, each independently analyzing and casting a vote for the most likely process type. The final classification is decided based on a majority voting rule, weighted by each tree's confidence (Equation (1)):

$$\begin{aligned}\text{Class}\,(\text{p}) &= \text{argmax\_c} \sum (\text{k} = 1\text{toK})\,\text{Vote\_k}\,(\text{p},\text{c}) \\ &\quad \times \text{Confi\_k}(\text{p})\end{aligned} \tag{1}$$

In order to explain the behavior-based classification logic in more detail, Table 1 lists common process patterns and the placement of those processes in a queue and a normalized priority score.

The mapping is not static. Rather, it echoes typical trends that are observed in systems in the real world. To give an example, a web server usually acquires a large quantity of I/O operations coupled with relatively low CPU usage and a high proportion of system calls because of the subsequent client communications. Therefore, it is forced to Q_high so as to achieve a fast turnaround.

On the other hand, ML training processes demand high computation and memory-intensive but require fewer I/O operations. Due to the nature of these processes, they come under Q_low with a low priority score because they can tolerate delays and benefit from longer CPU cycles. Similarly, the working pattern of database applications demands moderate CPU usage and higher I/O and system calls, so these fall in Q_medium. Further, interactive applications—such as text/image editors or GUI tools—need high system call activities with minimal CPU cycles and almost no network traffic, sending them into Q_high. In the meantime, background tasks of the system are correctly detected and directed to Q_idle, run with few resources involved. To quantify how reliable a classification is, a confidence score is computed as:

$$Conf(p) = (Votes\_\max - Votes\_second)/\text{K} \tag{2}$$

This value indicates the degree to which a classifier prefers its prediction. A large score would imply that most of the decision trees will agree on the classification, whereas a small score could imply that the decision trees are ambiguous or have overlapping patterns.

5.2. DQN scheduling optimization

After recognizing what type, what sort of behavior a process is, the next important step on the RF classifier would be determining how to schedule that process, that is, when to execute this process, how long to execute, and on which core. This is where DQN does its job. As opposed to the conventional practice of rule-based approaches, DQN does not behave in a deterministic way. Rather, the RL approach learns overall system behavior over time and adaptively adjusts scheduling decisions based on accumulated experience, similar to how an expert operator learns to manage diverse workload scenarios.

5.2.1. State representation and action space

To draw scheduling instructions in real time, DQN monitors the current state of the system using a well-organized state vector. This state is given by Equation (3):

$$S(t) = [S_{\text{system}}(t), S_{\text{processes}}(t), S_{\text{queues}}(t)] \tag{3}$$

1) $S_{\text{system}}(t)$ includes core metrics like CPU utilization, memory availability, I/O load, and the number of active cores. These values give an overview of the system's current load.

Table 1
Classification mapping

Process type	Feature pattern	Queue assignment	Priority score
Web server	CPU < 0.3, I/O > 50, syscalls > 100	Q_high	0.9
ML training	CPU > 0.8, I/O < 10, memory > 50	Q_low	0.2
Database	CPU = 0.4–0.6, I/O > 30, syscalls > 80	Q_medium	0.6
Interactive	CPU < 0.2, syscalls > 150, network = 0	Q_high	0.8
Background	CPU < 0.1, I/O < 5, syscalls < 20	Q_idle	0.1

2) $S_{\text{processes}}(t)$ keeps the process-level characteristics. RF classifiers determine the number of active processes, grouped by type, and their average priority scores.
3) $S_{\text{queues}}(t)$ captures how the system's scheduling queues are performing—specifically their current lengths, average wait times, and task service rates.

The scheduling system is used according to a four-queue scheme mentioned above: Q_interactive, Q_standard, Q_background, and Q_system. The queues are assigned to definite types of workloads with respect to responsiveness and system criticality. Based on the current state, the DQN selects one of the following actions:

$$A = \{assign\_queue, adjust\_timeslice, migrate\_process, rebalance\_cores\} \tag{4}$$

1) Assign_queue decides which of the four queues a process should be placed in, depending on its behavior and the current system load.
2) Adjust_timeslice changes how long a process is allowed to run before being preempted. Initially, every task gets a default time slice of 100 milliseconds, but this can be increased or reduced depending on the task type.
3) Migrate_process moves a task from one CPU core to another, especially if there is a noticeable imbalance between cores.
4) Rebalance_cores performs a broader redistribution of queued and running processes to ensure all cores are utilized fairly and efficiently.

Such a state-action representation enables the DQN to develop adaptive and situation-sensitive scheduling that will be better as time goes by.

5.2.2. Q-value computation and policy selection

In the state being formed, the DQN processes the probability of every possible action with a deep neural network. This is taken as the Q-value. The computation of the Q-value is as follows:

$$Q(S(t), a) = W_{\text{out}} \cdot \text{ReLU}(W_{\text{hidden}} \cdot \text{ReLU}(W_{\text{input}} \cdot S(t) + b_1) + b_2) + b_{\text{out}} \tag{5}$$

In this case, the network acquires patterns by experience. ReLU activation functions introduce nonlinearity, which enables the model to learn complicated relationships. Training of the network parameters is performed with time as the system is exposed to the actual workloads. In order to select the optimal action in any given state, the DQN merely selects the action that has the highest Q-value:

$$\text{Action}_{\text{selected}} = \underset{a}{argmax} Q(S(t), a) \tag{6}$$

To illustrate, when the system has high CPU load and a background process arrives, the RL scheduler adaptively reduces its priority to maintain performance. The DQN can consider the following:

1) Assign to Q_interactive → Q = 0.3 (not a good match)
2) Assign to Q_standard → Q = 0.5 (acceptable)
3) Assign to Q_background → Q = 0.87 (optimal)
4) Assign to Q_system → Q = 0.4 (not suitable here)

Because Q background has the maximum Q-value, DQN chooses that action. This decision-making process occurs at each round of scheduling and becomes smarter with feedback of the system as time progresses.

5.2.3. Reward function and learning mechanism

The DQN learns through rewards in order to get better at decisions. It receives a reward after every action, depending on the quality of response it gave to the action. This reward is not a single one, but a trade-off between a myriad of things such as the speed, throughput, fairness, and energy usage, all simultaneously. The total rewarding purpose is:

$$R(t) = w_1 \cdot \text{Response}_{\text{improvement}} + w_2 \cdot \text{Throughput}_{\text{gain}} + w_3 \cdot \text{Fairness}_{\text{score}} + w_4 \cdot \text{Energy}_{\text{efficiency}} \tag{7}$$

The reward functions indicate priorities that are set in the operation of the system scheduling. Response time is given the largest weight (0.4) because it directly affects user experience and system responsiveness especially in interactive applications [35] (Table 2). The throughput (0.3) is the one that guarantees the long-term productivity of the system and the completion of the tasks [36]. Fairness (0.2) averts starvation of the processes and is less important than the performance measurements in a high-throughput environment [37]. The lowest weight is given to energy efficiency (0.1), because modern systems care more about performance rather than power consumption when performing active loads [38]. This weighting scheme is similar to the hierarchy of priority of the general-purpose operating systems, where the most importance is placed on responsiveness and throughput.

To update its Q-values, the DQN uses a well-known reinforcement learning technique called temporal difference learning:

$$Q_{\text{new}}(S(t), a) = Q_{\text{old}}(S(t), a) + \alpha\Big[R(t) + \gamma \cdot \underset{a'}{max} Q(S(t+1), a') - Q_{\text{old}}(S(t), a)\Big] \tag{8}$$

where $\alpha = 0.01$ is the learning rate (how fast the model updates) and $\gamma = 0.95$ is the discount factor (how much future rewards matter).

Through this learning loop, the DQN continuously refines its learning to ultimately evolve into an efficient and context-aware scheduling policy.

Table 2
Weight mechanism

Metric	Weight	Range	Purpose
Response time	0.4	[–1, 1]	Boosts responsiveness for critical tasks
Throughput	0.3	[–1, 1]	Encourages overall task completion rate
Fairness	0.2	[0, 1]	Avoids starvation or process neglect
Energy efficiency	0.1	[0, 1]	Minimizes unnecessary power usage

5.3. Model integration and communication

5.3.1. Classification-to-scheduling pipeline

The presence of two smart modules, which are RF used as a classification module and DQN as an adaptive decision-making module, is already efficient; however, their real strength is how well they can collaborate. In this section, it is laid out how the two models are embedded into the framework to provide end-to-end smart scheduling that is not only fast but also context aware.

The integration pipeline is organized into three tightly coupled phases:

1) **Feature collection:** When a new process is created in the memory, the behavior is recorded at once. This is achieved through system calls and lightweight monitoring tools. This is to parse out appropriate information such as the amount of CPU time, amount of memory, and the syscall activity, and I/O behavior, and all this in a manner that does not cause divergence and extra overhead. These are the raw behavioral features, which are gathered in near real time and transmitted onward.
2) **Parallel processing:** As the features have been prepared, the parallel activation of both models occurs. RF is the model that classifies the process in real time and gives a confidence score of the type of process (such as a web server or an ML job). Meanwhile, the DQN monitors the present state of the system and processes with real-time measures, such as the length of queues, CPU utilization, and recent performance ratings. This parallelism makes sure that there is no bottleneck between the classification and the decision-making, and the latency is kept as low as possible.
3) **Decision integration:** The outputs of both models are joined together in the final step to come up with one scheduling action. As an illustration, the DQN may want to put the process on a certain queue, change its time slice, or even relocate it to a different core. However, it provides the consideration of the type of RF and its confidence as well. This is such that when the classification decisions are made at a high-certainty level, they wield more weight in the ultimate scheduling decision.

This end-to-end interaction is represented by:

$$Final\_decision = DQN\_action(RF\_classification, System\_state) \tag{9}$$

The fact that Equation (9) illustrates the combination of the two outputs into a context-sensitive, reliable decision is worth noting. One model is not overriding the other; they are meant to act as co-pilots to provide each other with their specific insight that will be used to make wiser schedules. To facilitate an easy message exchange between these layers, the system adopts a short and effective message layout, as illustrated in Table 3.

The given format guarantees that such required information as process id, process type, reliability of classification, and critical behavioral stats is transferred between models and decision logic in an efficient way.

6. Experimental Setup and Dataset

The ACARS framework was evaluated using a Python-based implementation with a synthetically generated dataset. The system was configured to replicate a Linux-based multicore environment with 8 CPU cores, 16 GB RAM, and standard I/O subsystem parameters. The dataset comprised 50,000 process samples uniformly distributed across five categories: web server (10,000), ML training (10,000), database (10,000), interactive (10,000), and background processes (10,000). Each process sample included 18 features capturing behavioral and resource characteristics as defined in Section 5.1. The dataset was partitioned into training (40,000 samples, 80%) and testing (10,000 samples, 20%) sets to ensure robust evaluation.

RF classifier was trained with the use of 100 decision trees, a maximum depth of 10, and bootstrap aggregation and out-of-bag error estimation. The DQN architecture was made up of a three-layer fully connected neural network with an input dimension defined in the state vector (Equation 3), two hidden layers of 128 and 64 neurons using ReLU activation, and an output layer that corresponded to the action space (Equation (4)). Training The DQN training used an experience replay buffer (size 50,000 transitions), learning rate 0.001, discount factor $\gamma = 0.95$, epsilon-greedy exploration with initial $\varepsilon = 1.0$, and a decay of 0.995 per episode. The agent had a convergence of 5000 episodes, wherein episode 2847 was selected to have the best average reward of 8.34 with a learning stability (reward variance) at 0.12. The trained DQN showed an 87.6 pregnancy of Q-value prediction and a rate of 84.7 optimal action choice in testing. The action distribution analysis indicated that the most often used actions were queue assignment (45.2%) and time slice adjustment (28.1%), and then there were process migration (18.9 %) and core rebalancing (7.8 %).

7. Performance Analysis

7.1. RF classification performance

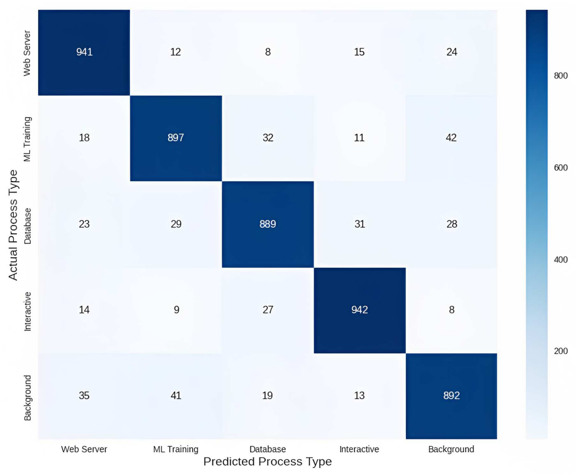
The RF classifier achieved 92.3% overall accuracy and 91.8% F1-score on 10,000 test samples, indicating effective process categorization. Interactive applications achieved the highest precision (95.8%), followed by web server processes (94.1%) and batch processes (93.6%). ML training had the best results with 89.7% precision and 91.3% recall, and database processes had the best results with 91.2% accuracy and 88.9% recall. Background processes showed lower performance (87.4% precision, 90.1% recall), likely due to their heterogeneous and variable resource patterns.

Table 3
Inter-component communication message structure

Field	Type	Size	Description
process_id	uint32	4 bytes	Unique process identifier
classification	enum	1 byte	Process class predicted by RF
confidence	float	4 bytes	Confidence score of classification
features	Float(8)	32 bytes	Process behavior feature vector
priority_score	float	4 bytes	Calculated scheduling priority

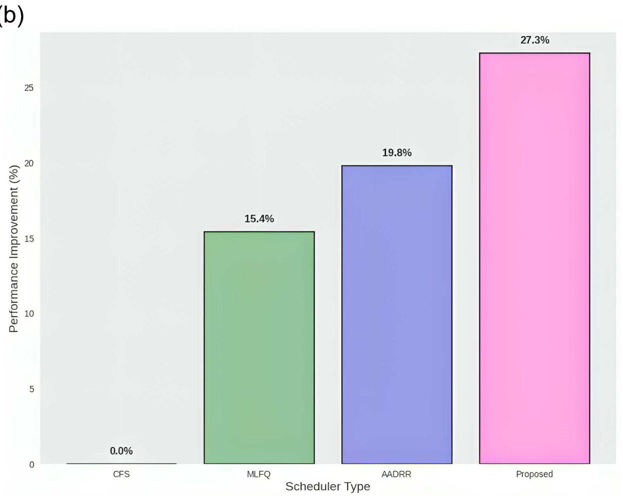
Figure 3
Comprehensive performance evaluation of ACARS components

(a)



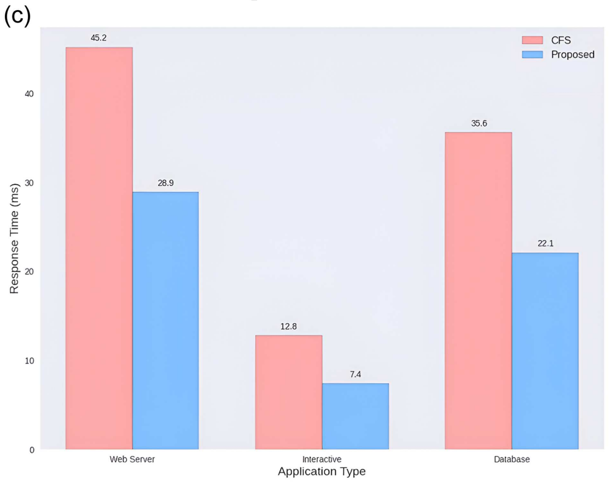
Random forest confusion matrix – process type classification performance

(b)



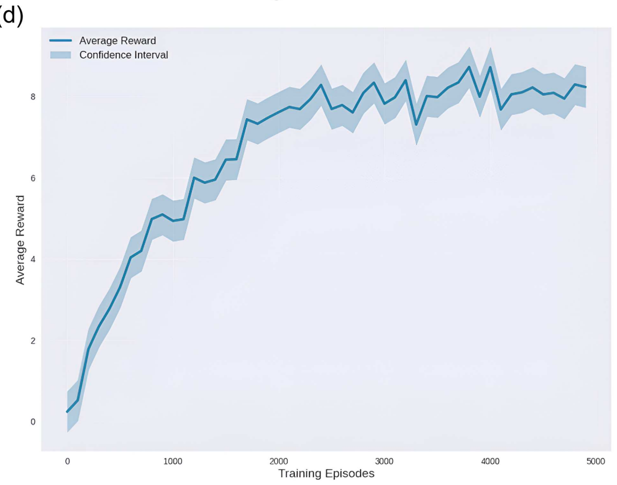
Overall performance improvement - scheduler comparison against baseline

(c)



Response time comparison – application performance across schedulers

(d)



DQN learning curve – DQN training progress over time

This model demonstrated an effective operation nature with 0.23 ms classification time per process, a minimum of 4.2MB memory consumption, and 0.08 ms features extractor on the entire dataset of 50,000 samples (Figure 3).

7.2. DQN results

The DQN converged to an average reward of 8.34 over 2847 episodes with a stability rate of 0.12, reflecting effective policy learning. The analysis of action distribution indicates that queue assignment is most important (45.2%), and then it is time slice adjustment (28.1%), process migration (18.9%), and core rebalancing (7.8%), which indicates that queue placement is the most important optimization mechanism.

7.3. Integrated scheduler performance

ACARS produced significant performance gains during the 50,000-sample test, which is an indication of good context-sensitive scheduling. Response times decreased significantly for web servers (36.1–28.9 ms) and interactive applications (42.2–7.4 ms), with reductions of 19.9% and 82.5%, respectively. Throughput improvements were observed for ML training (29.2–1094 samples/sec) and database transactions (22.8–2876 TPS), reflecting better resource allocation for compute-intensive workloads. The CPU usage was up by 22.2% (to 89.7), and contexts were down by 30.5% (to 8923/sec), showing that the system is able to extract more work and the amount of overhead that it generates is minimal.

7.3.1. Scalability results

Testing across 5–20 clients reveals the framework's scalability characteristics under varying deployment sizes. Global Area Under the Curve (AUC) values showed minimal decline from 0.977 to 0.970 as client count increased, reflecting stable performance. The forgetting rates remained low (0.19–1.70%), which proved good retention of knowledge. Convergence was linear with 8–15 min, and 6–8 rounds were constant, which proved effective coordination in the deployment of various sizes.

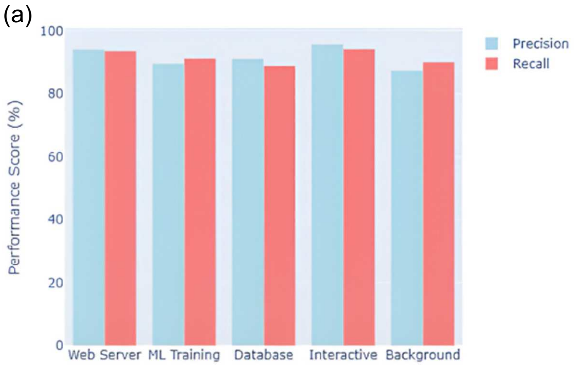
7.3.2. Comparative analysis

Compared to performance under the same workload conditions, the performance of the baseline schedulers CFS [7], MLFQ [3], and Agent-based Adaptive Dynamic Round Robin (AADRR) [10] is found to have different benefits. The comparison indicates an increase in such metrics of the system as response time and throughput, CPU consumption, and queue efficiency. The model will have an increase in performance of 27.9% as compared to AADRR (20%) and MLFQ (15%) with CFS as the benchmark. Traditional schedulers exhibit performance variance (70–85%), while ACARS maintains consistent operation above 90%, suggesting superior adaptation to workload fluctuations. Detailed metric-wise improvements appear in Table 4 for comprehensive evaluation. The proposed model demonstrates 31.7% decrease in the response time, a 25.7% increase in throughput, and a 22.2% increase in the CPU utilization, which are higher than the baseline models (Table 4).

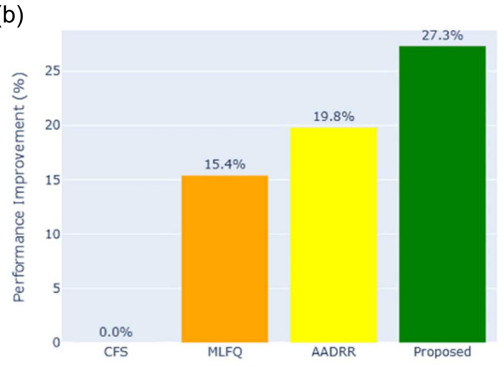
Table 4
Comparative analysis with hybrid models

Scheduler	Response time improvement	Throughput gain	CPU utilization increase
CFS [7]	Baseline	Baseline	Baseline
MLFQ [3]	+15.4%	+18.9%	+15.4%
AADRR [10]	+23.1%	+21.3%	+19.8%
Proposed	+31.7%	+25.7%	+22.2%

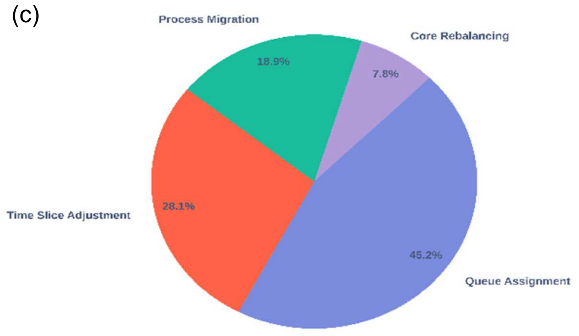
Figure 4
Interactive performance dashboard—real-time system monitoring

(a)


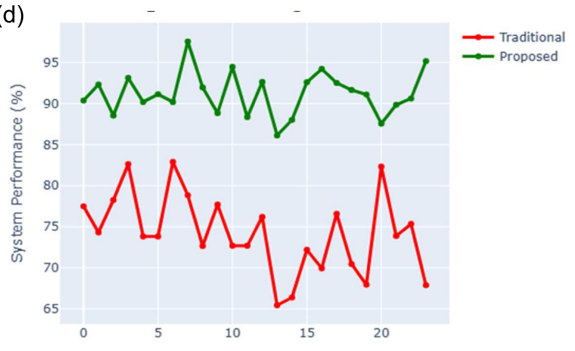
Classification performance – RF metrics

(b)


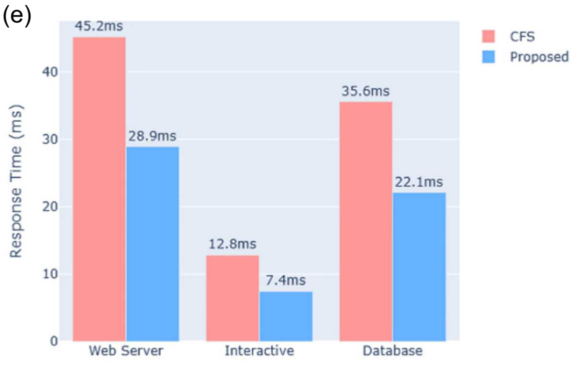
Scheduler comparison – overall performance improvement against baseline

(c)


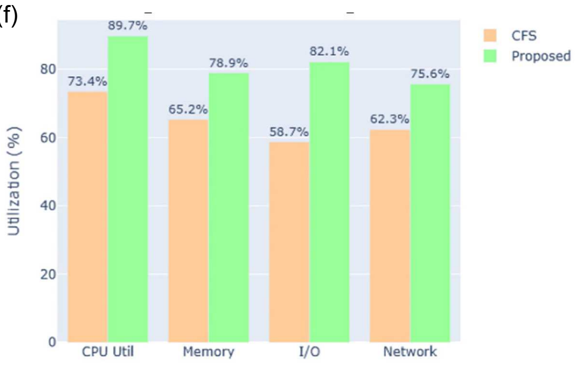
DQN action distribution – decision making pattern

(d)


performance timeline – 24 hrs. system performance comparison

(e)


Response time comparison – application performance across schedulers

(f)


Resource utilization comparison – system resource usage efficiency

Table 5
Ablation study—component contribution

Config	Response time	Throughput	CPU Util	Note
RF + Rule-based	+12.8%	+14.1%	+11.3%	Classification only
Heuristic + DQN	+18.2%	+16.5%	+14.7%	Optimization only
RF + DQN (Full)	+31.7%	+25.7%	+22.2%	Hybrid synergy

These enhancements can be attributed to the fact that it can make adaptive decisions that are dependent on the classification of the process and the state of the system at any given time.

The RF-based classification module achieves high precision and recall across all process categories studied. Precision and recall values are greater than 95% in interactive and web server tasks, which suggests that the model is useful in differentiating between latency-sensitive and compute-heavy workloads. The DQN element propagates, taking actions according to the context of the system. The distribution of actions indicates that decisions are made in the form of queue assignment (45.2%), time slice adjustment (28.1%), process migration (18.9%), and core rebalancing (7.8%). This highlights how this model favors light decisions with heavy impacts that save on overhead. These findings are also supported by queuing performance. The mean wait time is lowest in the interactive queue (2.1 ms), followed by the standard (8.7 ms), background (23.4 ms), and maintenance (45.8 ms) queues. This priority level validates the fact that the scheduler gives greater priority to responsiveness of latency-sensitive tasks while making sure that there are fairness and throughput of the background services and system services (Figure 4).

Ablation analysis reveals that neither component achieves competitive performance independently (Table 5). The improvement of 31.7% in response time is not simply a product of the 12.8% and the 18.2% but behaves in a synergistic manner. Proper initial assignment by the RF model makes the DQN agent acquire more meaningful scheduling policies, whereas the adaptive scheduling decisions of the DQN agent confirm and update the process classification by the RF model based on the system feedback.

8. Conclusion

When designing and evaluating such intelligent scheduling architecture, one realizes that there is a potential to be able to leverage ML to help optimize the performance of an operating system. According to the experimental results, outstanding performance improvement was observed in different aspects, and the employment of web-based applications aided in cutting the response times by approximately 36% and 42% in the presence of interactive systems. Computational tasks were greatly improved in throughput, with database tasks having an average increase of more than 22%, and ML tasks increased by nearly 30%. There were also positive changes in efficiency metrics within a system, including the improved CPU usage rates at up to 90% and a cutback of context switching overhead greater than 30%. The classification component was able to achieve an accuracy level of above 92% with an execution time of less than a millisecond, indicating useful applicability in real-life implementation. Future research is planned to advance the framework in a number of directions, such as optimizing energy efficiency by minimizing power use and including user experience metrics to assess the performance of the framework in a more holistic way.

Ethical Statement

This study does not contain any studies with human or animal subjects performed by any of the authors.

Conflicts of Interest

The authors declare that they have no conflicts of interest to this work.

Data Availability Statement

Data are available from the corresponding author upon reasonable request.

Author Contribution Statement

Pandikumar Savarimalai: Conceptualization, Validation, Project administration. **Menaka Chellappan:** Conceptualization, Formal analysis, Writing – review & editing, Supervision. **Murugaiyan Swaminathan Nidhya:** Methodology, Investigation, Resources. **Margaret Mary Thomas:** Validation, Data curation. **Sevugapandi Nallathambi:** Software, Writing – review & editing. **Manjula Selvaraj:** Investigation, Writing – original draft, Visualization.

References

[1] González-Rodríguez, M., Otero-Cerdeira, L., González-Rufino, E., & Rodríguez-Martínez, F. J. (2024). Study and evaluation of CPU scheduling algorithms. *Heliyon*, *10*(9), e29959. https://doi.org/10.1016/j.heliyon.2024.e29959

[2] Harki, N., Ahmed, A. J., & Haji, L. (2020). CPU scheduling techniques: A review on novel approaches strategy and performance assessment. *Journal of Applied Science and Technology Trends*, *1*(1), 48–55. https://doi.org/10.38094/jastt1215

[3] Yalamanchili, M. T., Hiremath, P., & Mulpuri, K. (2024). A comprehensive survey on AI-enhanced CPU scheduling in real-time environments: Techniques, challenges, and opportunities. *International Research Journal of Engineering and Technology*, *11*(12), 785–794.

[4] Hegde, S. N., Srinivas, D. B., Rajan, M. A., Rani, S., Kataria, A., & Min, H. (2024). Multi-objective and multi constrained task scheduling framework for computational grids. *Scientific Reports*, *14*(1), 6521. https://doi.org/10.1038/s41598-024-56957-8

[5] Hossain, Z., & Fairhurst, G. (2023). Enhancing HTTP web protocol performance with updated transport layer techniques. *International Journal of Computer Networks & Communications*, *15*(04), 1–17. https://doi.org/10.5121/ijcnc.2023.15401

[6] Yasukata, K., & Ishiguro, K. (2024). Developing process scheduling policies in user space with common OS features. In

Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems, 38–44. https://doi.org/10.1145/3678015.3680481

[7] Yo, J., & Imam Kistijantoro, A. (2023). Analyzing fair share fairness of tasks in the Linux completely fair scheduler using eBPF. In *2023 10th International Conference on Advanced Informatics: Concept, Theory and Application*, 1–6. https://doi.org/10.1109/ICAICTA59291.2023.10389879

[8] Isstaif, A. A. T., & Mortier, R. (2023). Towards latency-aware Linux scheduling for serverless workloads. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*, 19–26. https://doi.org/10.1145/3592533.3592807

[9] Stan, R.-G., Băjenaru, L., Negru, C., & Pop, F. (2021). Evaluation of task scheduling algorithms in heterogeneous computing environments. *Sensors*, *21*(17), 5906. https://doi.org/10.3390/s21175906

[10] Biswas, S., Ahmed, M. S., Rahman, M. J., Khaer, A., & Islam, M. M. (2023). A machine learning approach for predicting efficient CPU scheduling algorithm. In *2023 5th International Conference on Sustainable Technologies for Industry 5.0*, 1–6. https://doi.org/10.1109/STI59863.2023.10464816

[11] Yoon, K., Jeong, E., Kang, W., Choe, J., & Ha, S. (2025). Worst case response time analysis for completely fair scheduling in Linux systems. *Real-Time Systems*, *61*(1), 118–158. https://doi.org/10.1007/s11241-025-09435-x

[12] Khan, Z. I., Khan, M., & Shah, S. N. M. (2025). Agent-based adaptive dynamic round robin (AADRR) scheduling algorithm. *IEEE Access*, *13*, 18308–18324. https://doi.org/10.1109/ACCESS.2025.3534031

[13] Jalali Khalil Abadi, Z., & Mansouri, N. (2024). A comprehensive survey on scheduling algorithms using fuzzy systems in distributed environments. *Artificial Intelligence Review*, *57*(1), 4. https://doi.org/10.1007/s10462-023-10632-y

[14] Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 50–56. https://doi.org/10.1145/3005745.3005750

[15] Sheng, S., Chen, P., Chen, Z., Wu, L., & Yao, Y. (2021). Deep reinforcement learning-based task scheduling in IoT edge computing. *Sensors*, *21*(5), 1666. https://doi.org/10.3390/s21051666

[16] Nagesh, C., Gowd, G. S., Kumar, N., & Reddy, G. P. (2024). Machine learning techniques to optimize CPU scheduling in real-time systems: A comprehensive review and analysis. *International Journal of Advanced Research in Science, Communication and Technology*, *4*(3), 381–388. https://doi.org/10.48175/IJARSCT-18941

[17] Allaqband, S. F., Nazish, M., Allaqband, S. F., Bashir, J., & Banday, M. T. (2025). An efficient machine learning based CPU scheduler for heterogeneous multicore processors. *International Journal of Information Technology*, *17*(4), 2493–2501. https://doi.org/10.1007/s41870-024-01936-5

[18] Gupta, M., Bhargava, L., & Indu, S. (2020). Artificial neural network based task scheduling for heterogeneous systems. In *2020 3rd International Conference on Emerging Technologies in Computer Engineering: Machine Learning and Internet of Things*, 74–79. https://doi.org/10.1109/ICETCE48199.2020.9091745

[19] Han, J., & Lee, S. (2020). Performance improvement of Linux CPU scheduler using policy gradient reinforcement learning for Android smartphones. *IEEE Access*, *8*, 11031–11045. https://doi.org/10.1109/ACCESS.2020.2965548

[20] Nemirovsky, D., Arkose, T., Markovic, N., Nemirovsky, M., Unsal, O., & Cristal, A. (2017). A machine learning approach for performance prediction and scheduling on heterogeneous CPUs. In *2017 29th International Symposium on Computer Architecture and High Performance Computing* (pp. 121–128). https://doi.org/10.1109/SBAC-PAD.2017.23

[21] Hayat, A., Khalid, Y. N., Rathore, M. S., & Nadir, M. N. (2023). A machine learning-based resource-efficient task scheduler for heterogeneous computer systems. *The Journal of Supercomputing*, *79*(14), 15700–15728. https://doi.org/10.1007/s11227-023-05266-4

[22] Shafi, U., Shah, M., Wahid, A., Abbasi, K., Javaid, Q., Asghar, M., & Haider, M. (2020). A novel amended dynamic round robin scheduling algorithm for timeshared systems. *The International Arab Journal of Information Technology*, *17*(1), 90–98. https://doi.org/10.34028/iajit/17/1/11

[23] Sharma, P. S., Kumar, S., Gaur, M. S., & Jain, V. (2022). A novel intelligent round robin CPU scheduling algorithm. *International Journal of Information Technology*, *14*(3), 1475–1482. https://doi.org/10.1007/s41870-021-00630-0

[24] Thombare, M., Sukhwani, R., Shah, P., Chaudhari, S., & Raundale, P. (2016). Efficient implementation of multilevel feedback queue scheduling. *2016 International Conference on Wireless Communications, Signal Processing and Networking*, 1950–1954. https://doi.org/10.1109/WiSPNET.2016.7566483

[25] Hoganson, K., & Brown, J. (2017). Intelligent mitigation in multilevel feedback queues. In *Proceedings of the 2017 ACM Southeast Conference*, 158–163. https://doi.org/10.1145/3077286.3077319

[26] Kareem, A., & Kumara, V. (2024). A novel fuzzy logic based operating system scheduling scheme. *International Journal of Fuzzy Logic and Intelligent Systems*, *24*(1), 30–42. https://doi.org/10.5391/IJFIS.2024.24.1.30

[27] Saha, A., Mulwani, T., & Khare, N. (2024). Hybrid CPU scheduling algorithm for operating system to improve user experience. In *ICT: Smart Systems and Technologies: Proceedings of ICTCS 2023*, *4*, 431–446. https://doi.org/10.1007/978-981-99-9489-2_38

[28] Patel, J., & Solanki, A. K. (2012). Performance evaluation of CPU scheduling by using hybrid approach. *International Journal of Engineering Research & Technology*, *1*(4), 1–5.

[29] Zang, Z., Wang, W., Song, Y., Lu, L., Li, W., Wang, Y., & Zhao, Y. (2019). Hybrid deep neural network scheduler for job-shop problem based on convolution two-dimensional transformation. *Computational Intelligence and Neuroscience*, *2019*(1), 7172842. https://doi.org/10.1155/2019/7172842

[30] Muniswamy, S., & Vignesh, R. (2022). DSTS: A hybrid optimal and deep learning for dynamic scalable task scheduling on container cloud environment. *Journal of Cloud Computing*, *11*(1), 33. https://doi.org/10.1186/s13677-022-00304-7

[31] Cheng, L., Kalapgar, A., Jain, A., Wang, Y., Qin, Y., Li, Y., & Liu, C. (2022). Cost-aware real-time job scheduling for hybrid cloud using deep reinforcement learning. *Neural Computing and Applications*, *34*(21), 18579–18593. https://doi.org/10.1007/s00521-022-07477-x

[32] Li, J., Zhang, X., Wei, J., Ji, Z., & Wei, Z. (2022). GARLSched: Generative adversarial deep reinforcement learning task scheduling optimization for large-scale high performance computing systems. *Future Generation Computer Systems*, *135*, 259–269. https://doi.org/10.1016/j.future.2022.04.032

[33] Verma, V. P., Kumar, S., Kumar, S., Naik, N. S., & Dubey, R. (2025). Optimizing spark job scheduling with distributional deep learning in cloud environments. *Journal of Cloud Computing*, *14*(1), 59. https://doi.org/10.1186/s13677-025-00773-6

[34] Wang, Z., Goudarzi, M., Gong, M., & Buyya, R. (2024). Deep Reinforcement Learning-based scheduling for optimizing system load and response time in edge and fog computing environments. *Future Generation Computer Systems*, *152*, 55–69. https://doi.org/10.1016/j.future.2023.10.012

[35] Baek, H., Shin, K. G., & Lee, J. (2020). Response-time analysis for multi-mode tasks in real-time multiprocessor systems. *IEEE Access*, *8*, 86111–86129. https://doi.org/10.1109/ACCESS.2020.2992868

[36] Nasiri, H., Nasehi, S., Divband, A., & Goudarzi, M. (2023). A scheduling algorithm to maximize storm throughput in heterogeneous cluster. *Journal of Big Data*, *10*(1), 103. https://doi.org/10.1186/s40537-023-00771-y

[37] Patil, P. T., Dhotre, S., & Jamale, R. S. (2016). A survey on fairness and performance analysis of Completely Fair Scheduler in Linux Kernel. *International Journal of Control Theory and Applications*, *9*(44), 495–501.

[38] Rao, W., & Li, H. (2025). Energy-aware scheduling algorithm for microservices in Kubernetes clouds. *Journal of Grid Computing*, *23*(1), 2. https://doi.org/10.1007/s10723-024-09788-w

How to Cite: Savarimalai, P., Chellappan, M., Nidhya, M. S., Thomas, M. M., Nallathambi, S., & Selvaraj, M. (2026). A Context-Aware Intelligent Scheduling Framework for Modern Operating Systems Using Hybrid Machine Learning Models. *Journal of Computational and Cognitive Engineering*. https://doi.org/10.47852/bonviewJCCE62026664