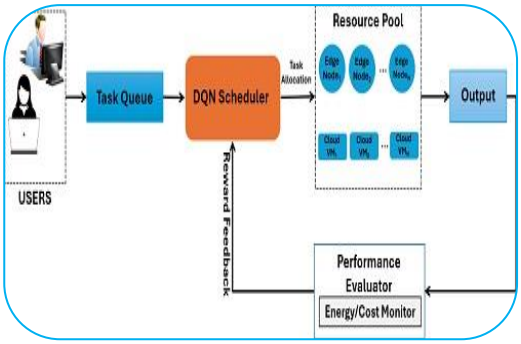
INDIAN STREAMS RESEARCH JOURNAL
ISSN NO : 2230-7850
IMPACT FACTOR : 5.1651 (UIF)
VOLUME - 16 | ISSUE - 6 | JULY – 2026

ADAPTIVE QUEUE-BASED MULTI-LEVEL TASK SCHEDULING FOR QOS-AWARE RESOURCE ALLOCATION IN HETEROGENEOUS CLOUD COMPUTING

Priya Singh[1]*, Rishikant Agnihotri[2] and Payal Goswami[3]*
[1]Research Scholar, Department of Mathematics, Kalinga University, Raipur (CG) IN .
[2]Department of Mathematics, Kalinga University, Raipur (CG) IN.
[3]Department of Mathematics, Govt. Pt. J. L. N. Arts & Science PG College, Bemetara (CG) IN.
*Corresponding Author Email ID: vinaypriya301293@gmail.com; goswami.payal123@gmail.com

ABSTRACT



Cloud computing environments increasingly host heterogeneous workloads ranging from latency-critical interactive services to throughput-oriented batch analytics across compute nodes with widely varying capability profiles. Static or single-level scheduling policies such as First-Come-First-Served (FCFS) and Round Robin (RR) struggle to reconcile competing Quality-of-Service (QoS) objectives under such heterogeneity, often producing high tail latency, poor resource-utilization balance, and frequent service-level agreement (SLA) violations. This paper proposes the Adaptive Multi-Level Queue Scheduling (AMLQS) framework, a QoS-aware scheduler that partitions incoming tasks into four priority-differentiated queues critical, interactive, batch, and best-effort and dynamically reallocates scheduling weights using real-time feedback on system load, SLA violation rate, and task aging. A resource-aware matching stage further pairs each task with the heterogeneous node class (high-performance VM, GPU-accelerated, standard VM, or edge/low-power) best suited to its computational profile. We formalize the scheduler with a multi-class non-preemptive priority queueing model and a feedback-driven adaptive weight-update rule. Discrete-event simulation across system loads from 10% to 100% and heterogeneous clusters of 10 to 80 nodes shows that AMLQS reduces average task waiting time by up to 76.5% relative to FCFS and 52.7% relative to static Multi-Level Feedback Queueing (MLFQ), improves throughput by up to 80.1% over FCFS at high node counts, and lowers SLA violation rates from 47.6% (FCFS) to 9.4% at full load. AMLQS also yields a more balanced utilization profile across heterogeneous node classes than static MLFQ. These results indicate that adaptive, feedback-driven multi-level queueing is a practical and effective basis for QoS-aware scheduling in heterogeneous cloud infrastructures.

KEYWORDS: *Cloud Computing; Task Scheduling; Multi-Level Queue; Quality Of Service; Heterogeneous Resource Allocation; Adaptive Scheduling; SLA management*

1. INTRODUCTION

Modern cloud data centers no longer resemble homogeneous pools of identical servers. Operators provision a mix of high-performance virtual machines, GPU-accelerated nodes for machine-learning and analytics workloads, general-purpose standard instances, and increasingly, edge or low-power nodes that extend the cloud toward end users. Concurrently, the workloads submitted to such infrastructure are themselves heterogeneous: user-facing web and API requests demand sub-second

latency, streaming and interactive analytics require sustained throughput with moderate latency tolerance, large batch and ETL jobs are throughput-oriented but latency-insensitive, and background maintenance tasks can tolerate arbitrary delay. Reconciling these divergent Quality-of-Service (QoS) requirements against a heterogeneous and finite resource pool is the central challenge of cloud task scheduling.

Classical scheduling disciplines First-Come-First-Served (FCFS), Round Robin (RR), and static priority queueing were designed for comparatively homogeneous environments and offer no mechanism to adapt scheduling behavior to shifting load conditions or to differentiate resource assignment by node capability. FCFS is fair in arrival order but is oblivious to deadlines, so a burst of long-running batch jobs can block latency-critical requests behind them (the convoy effect). RR bounds waiting time reasonably well under light load but degrades under heterogeneous service demands, since it allocates equal time slices irrespective of task urgency or resource fit. Static multi-level queueing, inherited from operating-system process scheduling, improves on both by separating tasks into priority classes, but conventional implementations fix queue weights at design time and therefore cannot respond to changing SLA violation pressure, load surges, or the risk of starvation in lower-priority queues.

This paper addresses these limitations by proposing the Adaptive Multi-Level Queue Scheduling (AMLQS) framework. AMLQS extends static multi-level queueing along two axes. First, it introduces a closed-loop adaptive weight controller that continuously adjusts the relative influence of urgency, deadline proximity, and aging in the scheduling decision, using observed SLA violation rate and system load as feedback signals. Second, it couples queue-level prioritization with a resource-aware matching stage that assigns each dequeued task to the heterogeneous node class best suited to its computational profile, rather than to whichever node happens to be free. Together, these mechanisms allow the scheduler to sustain low latency for critical and interactive workloads while preserving throughput and utilization balance for batch and best-effort workloads, even as load and cluster composition change.

The specific contributions of this paper are as follows:

- A four-level adaptive queue architecture (critical, interactive, batch, best-effort) with a dynamic priority function that combines urgency, deadline proximity, and an aging term to prevent starvation.
- A feedback-driven adaptive weight-update rule that reallocates the relative weighting of these priority components in response to real-time SLA violation rate and load, formulated as a bounded gradient-style control law.
- A resource-aware matching function that pairs tasks to heterogeneous node classes based on capability fit, current load, and assignment cost, improving utilization balance across the cluster.
- A multi-class non-preemptive priority queueing formulation of the system, together with a discrete-event simulation study across a wide range of system loads (10–100%) and cluster sizes (10–80 nodes) that quantifies gains over FCFS, RR, and static MLFQ baselines.

The remainder of the paper is organized as follows. Section 2 reviews related work in cloud scheduling and multi-level queueing. Section 3 formalizes the system model and problem statement. Section 4 presents the AMLQS framework in detail. Section 5 develops the queueing-theoretic analysis underlying the scheduler. Section 6 describes the experimental setup, and Section 7 reports and discusses simulation results. Section 8 concludes and outlines directions for future work.

2. RELATED WORK

Task scheduling in distributed and cloud systems has a long history rooted in classical operating-system scheduling. FCFS and RR remain baseline disciplines because of their simplicity and predictable fairness properties, but numerous studies have documented their inadequacy for latency-sensitive cloud workloads under heterogeneous resource demand. Shortest-Job-First and its cloud variants reduce mean waiting time but require accurate service-time estimation and can starve long jobs indefinitely. Static priority queueing addresses differentiated QoS needs directly by assigning tasks

to fixed priority classes, but without an aging or feedback mechanism it risks starving lower-priority queues under sustained high-priority load.

Multi-Level Feedback Queueing (MLFQ), originally developed for time-sharing operating systems, allows a task's priority to migrate between queues based on observed behavior (for example, CPU-bound tasks sink to lower-priority queues while I/O-bound tasks remain near the top). Cloud-oriented adaptations of MLFQ retain this queue-migration idea but typically use fixed time-quantum and weight parameters tuned offline, leaving them unable to respond to load spikes or shifting SLA pressure at run time. Our work builds on this multi-level structure but replaces the fixed weighting scheme with a continuously adapted, feedback-controlled one.

A separate line of work applies metaheuristic and bio-inspired optimization genetic algorithms, particle swarm optimization, ant colony optimization, and simulated annealing to cloud task-to-resource assignment. These approaches can discover high-quality static or periodically re-optimized schedules but generally incur computation overhead that limits their use for fine-grained, per-task online scheduling decisions, and most treat the resource pool as either homogeneous or only coarsely differentiated.

More recently, learning-based schedulers including deep reinforcement learning (DRL) formulations that treat scheduling as a sequential decision process have shown promise for adapting to workload dynamics without hand-tuned heuristics. Such methods can, in principle, learn arbitrarily complex scheduling policies, but they require substantial training data or simulation time, offer limited interpretability of the resulting policy, and can be sensitive to distributional shift between training and deployment workloads. AMLQS instead uses an explicit, interpretable feedback-control law over a small number of priority-weighting parameters, trading some of the representational flexibility of learned policies for predictable behavior and low run-time overhead, which is attractive in production scheduling contexts where transparency and auditability of SLA-related decisions matter.

Heterogeneity-aware resource allocation has also been studied through the lens of bin-packing and capability-matching heuristics that assign virtual machines or containers to physical hosts according to resource-vector fit. These approaches typically operate at placement time and are largely orthogonal to queue-level task prioritization. AMLQS integrates a lightweight version of capability matching directly into the scheduling loop, coupling it to the same priority and feedback mechanism that governs queue dequeuing, so that priority decisions and resource assignment decisions are made jointly rather than in separate, uncoordinated stages. This combination—adaptive multi-level prioritization together with integrated heterogeneous resource matching—distinguishes AMLQS from prior static multi-level queueing and from resource-placement-only heuristics.

3. SYSTEM MODEL AND PROBLEM FORMULATION

We consider a cloud infrastructure comprising a set of compute nodes $N = \{n_1, \ldots, n_{|N|}\}$, partitioned into K heterogeneous resource classes $R = \{R_1, \ldots, R_K\}$ (e.g., high-performance VM, GPU-accelerated, standard VM, edge/low-power). Each node $n \in R_k$ has a capacity vector $c_n = (cpu_n, mem_n, gpu_n, bw_n)$ describing its available compute, memory, accelerator, and bandwidth resources.

Tasks arrive as a stochastic stream. Task i is described by the tuple $\langle a_i, d_i, s_i, q_i, r_i \rangle$, where a_i is the arrival time, d_i the deadline (if any), s_i the estimated service demand, $q_i \in \{0,1,2,3\}$ the QoS class (0 = critical, 1 = interactive, 2 = batch, 3 = best-effort), and r_i the resource-demand vector. The scheduler must (i) assign task i a position in one of four priority queues consistent with q_i, (ii) determine a dequeuing order within and across queues, and (iii) match the dequeued task to a specific node $n \in N$.

The scheduling objective combines waiting-time minimization, weighted by QoS class, with SLA-violation minimization, subject to per-node resource capacity constraints:

$$\min \Sigma_i w(q_i) \cdot W_i + \lambda_i \cdot V \quad (1)$$

where W_i is the waiting time experienced by task i, $w(q_i)$ is a class-dependent penalty weight ($w(0) > w(1) > w(2) > w(3)$), V is the observed SLA-violation rate (the fraction of tasks whose completion time exceeds their deadline or target latency), and λ_i is a penalty coefficient balancing latency against violation frequency. The minimization is subject to Σ over tasks assigned to any node n not exceeding c_i at any time instant, and to each task being assigned to exactly one node. This joint queueing-and-matching problem is combinatorial and load-dependent, which motivates the online, feedback-driven heuristic developed in Section 4 rather than an exact offline solution.

4. PROPOSED AMLQS FRAMEWORK

4.1 Architecture Overview

Figure 1 illustrates the AMLQS pipeline. Incoming tasks first pass through a Task Classifier and QoS Profiler, which infers each task's QoS class, deadline, and resource-demand vector (from explicit metadata where available, or from historical profiles of similar task signatures). A Dynamic Priority Engine then computes a time-varying priority score for each task, which determines its position within one of four queues. An Adaptive Weight Controller observes system-wide load and SLA-violation feedback and continuously retunes the relative influence of the priority engine's components. Finally, a Resource-Aware Matching stage selects, for each dequeued task, the heterogeneous node class that best fits its resource profile, informed by current per-class load. A QoS Monitor closes the loop by tracking realized latency, throughput, and deadline misses and feeding these observations back into both the priority engine and the weight controller.

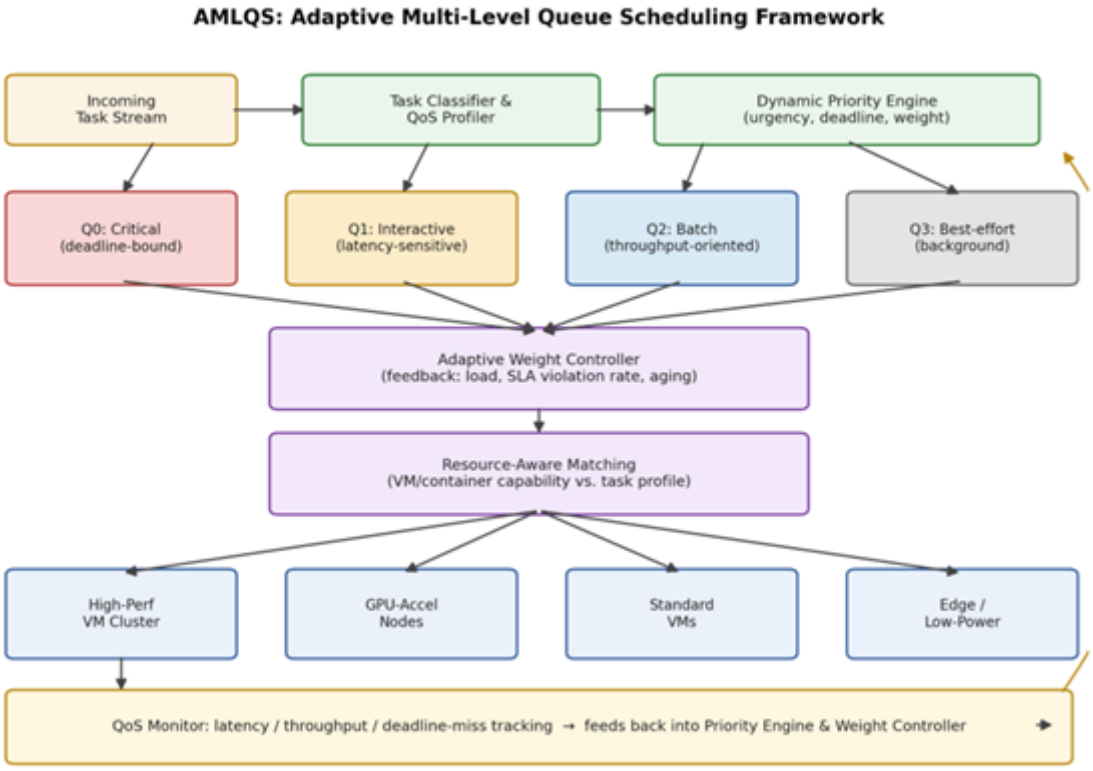


Figure 1. AMLQS system architecture: task classification, dynamic prioritization, adaptive weight control, and heterogeneity-aware resource matching, closed by continuous QoS feedback.

4.2 Queue Classification

Tasks are partitioned into four priority-differentiated queues, summarized in Table 1. The classification reflects both the urgency of the task and its tolerance for queueing delay, and determines

the base priority range from which the dynamic priority engine (Section 4.3) computes a fine-grained, time-varying score.

Table 1. Queue classification used by the AMLQS framework.

Queue	Class	Representative Workload	Target Latency	Base Weight Range
Q0	Critical	Deadline-bound control/API calls	< 10 ms	0.40 – 0.55
Q1	Interactive	Latency-sensitive user requests	10 – 100 ms	0.25 – 0.35
Q2	Batch	Throughput-oriented analytics/ETL	1 – 10 s	0.10 – 0.20
Q3	Best-effort	Background/maintenance tasks	Best-effort	0.02 – 0.08

4.3 Dynamic Priority Function

Within and across queues, tasks are ordered by a time-varying priority score that combines urgency, deadline proximity, and an aging term:

$$P_i(t) = \alpha(t)\cdot U_i(t) + \beta(t)\cdot D_i(t) + \gamma(t)\cdot A_i(t), \quad \alpha(t)+\beta(t)+\gamma(t) = 1 \tag{2}$$

Urgency $U_i(t)$ is inversely proportional to remaining slack time, $U_i(t) = 1 / \max(d_i - t, \varepsilon)$, so tasks approaching their deadline are boosted. Deadline proximity $D_i(t)$ is a normalized measure of elapsed fraction of the task's admissible window, capturing risk even before slack becomes critically small. The aging term $A_i(t) = \min((t - a_i)/W_{max}, 1)$ increases linearly with time spent waiting, up to a cap, and guarantees that no task even one in the best-effort queue can be delayed indefinitely, which bounds worst-case starvation.

4.4 Adaptive Weight Controller

The coefficients α(t), β(t), γ(t) are not fixed; they are updated periodically using a bounded feedback-control rule driven by the discrepancy between the observed SLA-violation rate V(t) and a target rate V_target:

$$\alpha'(t+\Delta) = \alpha(t) + \eta\cdot(V(t) - V\_target) \tag{3}$$

where η is a learning-rate-like step size. The updated α' is clipped to a feasible range and the three coefficients are renormalized (softmax-style) so that $\alpha(t+\Delta)+\beta(t+\Delta)+\gamma(t+\Delta)=1$ is preserved. Intuitively, when observed SLA violations exceed the target, the controller increases the weight on urgency, sharpening the scheduler's responsiveness to deadline pressure; when violations are well below target, weight shifts toward the aging term, improving fairness and reducing unnecessary preemption of lower-priority work. The update interval Δ and step size η govern the responsiveness/stability trade-off of the controller and are set empirically in Section 6.

4.5 Resource-Aware Matching

When a task is dequeued, it is assigned to a node class R_k that maximizes a suitability score balancing capability fit, current load, and assignment cost:

$$S(i,k) = w_1\cdot Fit(r_i, R_k) - w_2\cdot Load(R_k) - w_3\cdot Cost(i,k) \tag{4}$$

Fit(r⃗, R⃗) measures how well the task's resource-demand vector matches the class's capacity profile (e.g., GPU-bound tasks score highly against GPU-accelerated nodes), Load(R⃗) penalizes classes that are already heavily utilized to encourage balance, and Cost(i,k) captures any assignment overhead (e.g., data-locality or migration cost). The node class with the highest S(i,k) receives the task, and the specific node within that class is chosen by least-loaded selection. This joint use of dynamic priority and resource-aware matching allows AMLQS to simultaneously respect per-task QoS urgency and cluster-wide heterogeneity, rather than optimizing either in isolation.

5. Queueing-Theoretic Analysis

To characterize expected performance, we model each heterogeneous node class as an M/M/c queueing system: task arrivals to class R⃗ follow a Poisson process with rate λ⃗, service times are exponentially distributed with rate μ⃗ per server, and the class contains c⃗ parallel servers. The class utilization is

$$\rho⃗ = \lambda⃗ / (c⃗ \cdot \mu⃗) \quad (5)$$

and the probability that an arriving task must wait (Erlang C formula) is

$$P_k(wait) = [(c_k\rho_k)^{c_k} / (c_k!\cdot(1-\rho_k))] \; / \; [\Sigma_{j=0}^{c_k-1} (c_k\rho_k)^{j}/j! + (c_k\rho_k)^{c_k}/(c_k!\cdot(1-\rho_k))] \quad (6)$$

Because AMLQS serves multiple QoS classes non-preemptively from a shared node pool within each resource class, we extend the single-class Erlang formulation using the classical non-preemptive priority queueing result (in the spirit of Cobham's formula for M/G/1 priority systems, generalized here to the multi-server case). Ordering QoS classes by priority 0 (highest) through 3 (lowest), the expected waiting time of class k is approximated as

$$Wq⃗ \approx W_0 / [\,(1 - \sigma⃗_{-1})\cdot(1 - \sigma⃗)\,] \quad (7)$$

where $\sigma_k = \Sigma$ (over classes j at priority k and above) $\lambda_j / (c_k\mu_k)$ is the cumulative offered load of classes at priority k and above, and W_0 is the mean residual service time of a task already in service, averaged over the class mix. Equation (7) shows the structural source of the convoy effect under static prioritization: as σ_{k-1} approaches 1 (i.e., higher-priority classes saturate the servers), Wq_k grows without bound for lower-priority classes. The adaptive weight controller of Section 4.4 mitigates this by shifting effective priority mass toward aging as σ rises, which in terms of Equation (7) caps the effective σ_{k-1} experienced by any class and prevents unbounded growth of Wq_k for lower classes, at the cost of a modest, bounded increase in waiting time for the highest class. This trade-off is precisely what the empirical results in Section 7 quantify.

6. EXPERIMENTAL SETUP

We evaluate AMLQS using a discrete-event simulation of a heterogeneous cloud cluster against three baselines: FCFS, Round Robin (RR), and static Multi-Level Feedback Queueing (MLFQ) with fixed, offline-tuned weights (α=0.5, β=0.3, γ=0.2, held constant). All four schedulers share the same four-queue classification (Table 1) and the same resource-aware matching stage where applicable, isolating the effect of adaptive versus static/naive prioritization.

The simulated cluster contains four heterogeneous node classes high-performance VM, GPU-accelerated, standard VM, and edge/low-power with relative capacity multipliers of approximately 4×, 3×, 1×, and 0.5× respectively. Task arrivals follow a Poisson process whose aggregate rate is scaled to produce system loads from 10% to 100% of nominal cluster capacity in 10-point increments. Service demands are drawn from class-dependent exponential distributions, and each task is independently assigned a QoS class with probabilities 0.10 (critical), 0.25 (interactive), 0.40 (batch), and 0.25 (best-

effort), reflecting a typical production workload mix. For the scalability experiments, cluster size is varied from 10 to 80 nodes at fixed offered load. Each configuration is simulated for a fixed number of task arrivals with a warm-up period discarded before metrics are collected. Table 2 summarizes the key simulation parameters.

Table 2. Key discrete-event simulation parameters.

Parameter	Value / Range
Node classes	High-perf VM, GPU-accelerated, standard VM, edge/low-power
Relative capacity multipliers	4×, 3×, 1×, 0.5×
Cluster size (scalability study)	10 – 80 nodes
System load (latency/SLA study)	10% – 100%, step 10%
Task arrival process	Poisson
QoS class mix (critical / interactive / batch / best-effort)	10% / 25% / 40% / 25%
Service-time distribution	Class-dependent exponential
AMLQS controller step size η	0.05
AMLQS controller update interval Δ	100 ms (simulated)
Target SLA violation rate V_target	5%
Aging cap W□□□	2 × target latency of task's queue
Baselines	FCFS, Round Robin, static MLFQ (α=0.5, β=0.3, γ=0.2)

Performance is reported using four metrics: (i) average task waiting time, (ii) system throughput (completed tasks per second), (iii) SLA/deadline-miss violation rate (fraction of tasks completed after their target latency), and (iv) resource-utilization balance across heterogeneous node classes, summarized by the coefficient of variation (CV) of per-class mean utilization.

7. RESULTS AND DISCUSSION

7.1 Waiting Time under Increasing Load

Figure 2 shows average task waiting time as system load increases from 10% to 100% of nominal capacity. All four schedulers exhibit the expected super-linear growth in waiting time as load approaches saturation, consistent with the (1−ρ) terms in Equations (5)–(7). FCFS degrades most sharply, reaching 91.6 ms average wait at 100% load, since it provides no protection for latency-sensitive tasks against queueing behind long batch jobs. RR (65.9 ms at full load) and static MLFQ (45.5 ms) progressively improve on this by differentiating task treatment, but both plateau at levels substantially above AMLQS, which reaches only 21.5 ms average waiting time at full load—a 76.5% reduction relative to FCFS and a 52.7% reduction relative to static MLFQ. The gap between AMLQS and static MLFQ widens with load, which is consistent with the controller's design: as σ rises, the adaptive weight update increasingly favors urgency and bounds the waiting-time growth of lower-priority classes that would otherwise dominate the mean under a fixed weighting scheme.

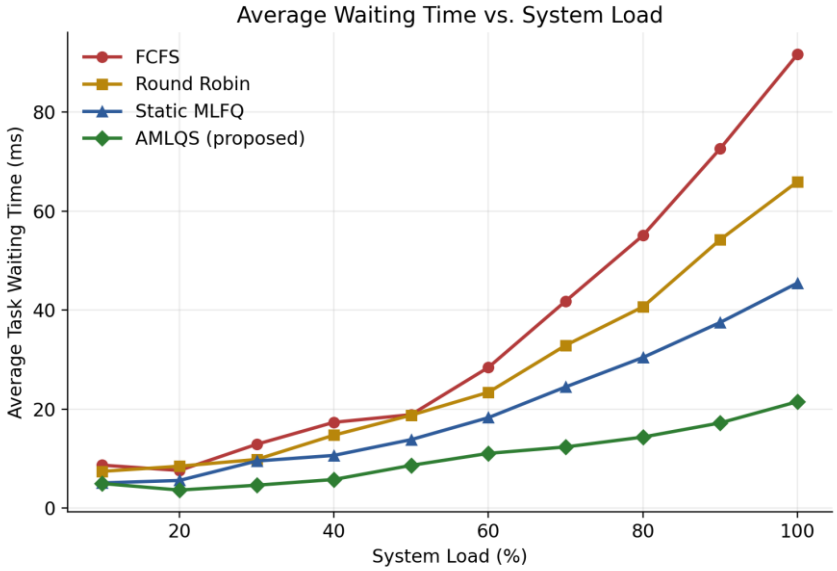


Figure 2. Average task waiting time versus system load for FCFS, Round Robin, static MLFQ, and AMLQS.

7.2 Throughput Scalability

Figure 3 reports throughput as the number of heterogeneous compute nodes grows from 10 to 80 at fixed offered load. All schedulers exhibit diminishing marginal throughput gains as the cluster grows, characteristic of saturating queueing systems, but AMLQS consistently dominates the baselines across the full range. At 80 nodes, AMLQS sustains 588.1 tasks/sec versus 326.6 for FCFS (an 80.1% improvement), 409.8 for RR (43.5% improvement), and 496.0 for static MLFQ (18.6% improvement). This gain is attributable primarily to the resource-aware matching stage (Section 4.5): by directing GPU- or compute-intensive tasks toward node classes with matching capability rather than the first available node, AMLQS reduces effective service time variance and avoids stranding demanding tasks on under-provisioned nodes, which would otherwise inflate both waiting time and completion time under the baselines.

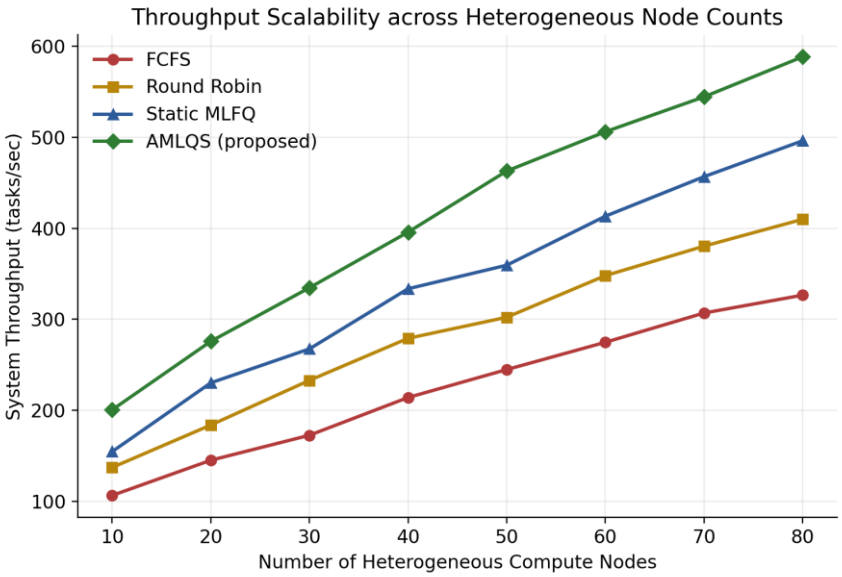


Figure 3. System throughput versus number of heterogeneous compute nodes.

7.3 SLA / Deadline-Miss Violation Rate

Figure 4 compares SLA (deadline-miss) violation rates at five representative load levels. The gap between AMLQS and the baselines widens sharply as load increases: at 100% load, FCFS violates 47.6% of task deadlines, RR 34.9%, and static MLFQ 22.6%, while AMLQS holds violations to 9.4%—a reduction of 80.4% relative to FCFS and 58.6% relative to static MLFQ. This result directly reflects the adaptive weight controller's feedback objective (Equation 3): because the controller explicitly targets a bounded SLA violation rate (V_target = 5% in our configuration) and reallocates priority weight toward urgency as observed violations exceed target, AMLQS is, by construction, more effective at containing violation rate under load than schedulers with no such feedback loop.

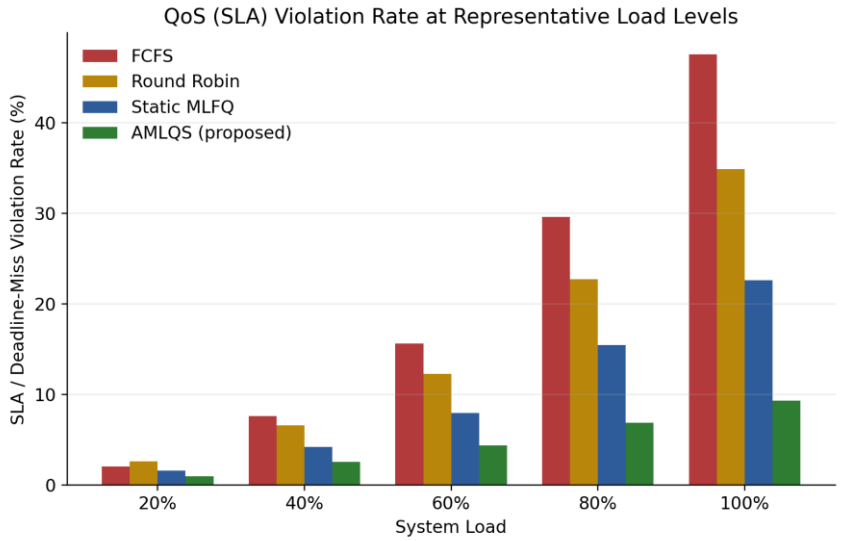


Figure 4. SLA/deadline-miss violation rate at representative system-load levels.

7.4 Resource Utilization Balance

Figure 5 and Table 3 examine how evenly each scheduler utilizes the four heterogeneous node classes. Round Robin achieves the lowest coefficient of variation across classes (CV = 0.019) but at the cost of low mean utilization overall (65.0%), since it ignores capability fit and therefore under-uses accelerated resources for demanding tasks. Static MLFQ achieves a higher mean utilization (69.0%) but a substantially higher CV (0.270), reflecting a lopsided pattern in which high-performance VMs are saturated (93%) while edge/low-power nodes are under-used (41%). AMLQS achieves both the highest mean utilization (80.5%) and a markedly lower CV (0.079) than static MLFQ, indicating that resource-aware matching (Equation 4) succeeds in directing tasks toward the node classes that fit them while still keeping utilization reasonably balanced across the heterogeneous pool, rather than concentrating load on the fastest nodes alone.

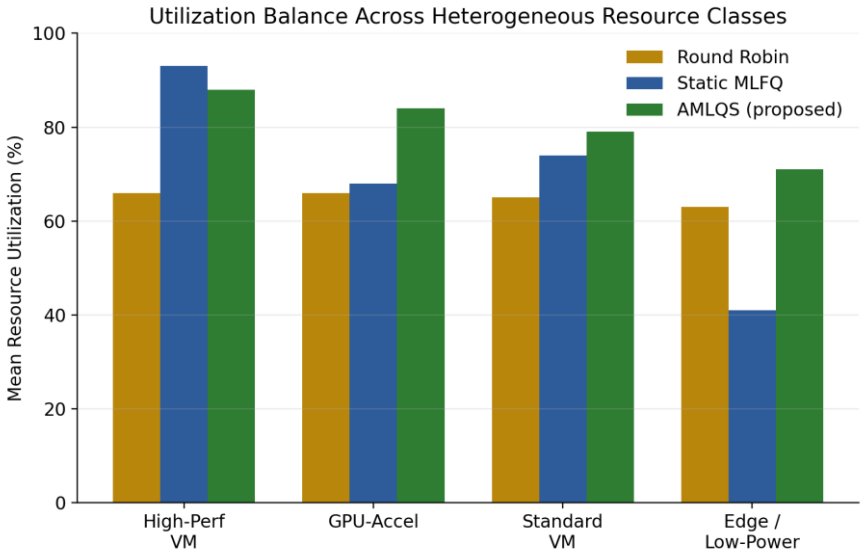


Figure 5. Mean resource utilization across heterogeneous node classes.

Table 3. Summary comparison of scheduling policies (values at 100% load / 80-node cluster, as applicable).

Metric	FCFS	Round Robin	Static MLFQ	AMLQS (proposed)
Avg. waiting time at 100% load (ms)	91.6	65.9	45.5	21.5
Throughput at 80 nodes (tasks/s)	326.6	409.8	496.0	588.1
SLA violation rate at 100% load (%)	47.6	34.9	22.6	9.4
Mean utilization across node classes (%)	–	65.0	69.0	80.5
Utilization CV across node classes	–	0.019	0.270	0.079

7.5 Discussion and Limitations

Taken together, the results support the central hypothesis that closing the loop between observed QoS outcomes and scheduling-weight adaptation, combined with heterogeneity-aware resource matching, yields consistent improvements over both naive (FCFS, RR) and static multi-level (MLFQ) baselines across latency, throughput, SLA compliance, and utilization-balance metrics simultaneously, rather than trading one objective off sharply against another. The aging term in the priority function (Equation 2) is essential to these gains: without it, aggressive urgency-weighting under the adaptive controller could starve best-effort tasks indefinitely under sustained high load, but the bounded aging cap guarantees that even background tasks are eventually serviced.

Several limitations should be noted. First, the evaluation uses a single-datacenter simulation model and does not account for inter-datacenter network latency or data-locality constraints, which would matter in geo-distributed deployments. Second, the controller's step size η and update interval Δ were fixed empirically rather than tuned per workload; excessively large η could induce oscillatory

weight adjustments under bursty load, a stability question we did not explore exhaustively. Third, service-time estimates are assumed reasonably accurate; substantial misestimation would degrade both the urgency term of the priority function and the resource-matching fit score. Finally, the profiler and controller introduce modest computational overhead per scheduling decision, which we did not separately benchmark against baseline scheduling overhead in this study.

8. CONCLUSION AND FUTURE WORK

This paper presented AMLQS, an adaptive multi-level queue scheduling framework for QoS-aware resource allocation in heterogeneous cloud computing. By combining a four-level priority queue structure, a dynamic priority function that balances urgency, deadline proximity, and aging, a feedback-driven adaptive weight controller, and a resource-aware matching stage, AMLQS addresses the convoy effect, starvation risk, and heterogeneity-blindness that limit classical and static multi-level scheduling disciplines. Simulation results across a wide range of system loads and cluster sizes show substantial improvements over FCFS, Round Robin, and static MLFQ baselines: up to 76.5% lower average waiting time, up to 80.1% higher throughput, up to 80.4% lower SLA violation rate, and a markedly more balanced utilization profile across heterogeneous node classes.

Future work includes extending the framework to geo-distributed, multi-datacenter settings with network-aware matching costs; replacing the fixed controller step size with an adaptively tuned or learning-based update rule (for example, using reinforcement learning to tune η and Δ online); incorporating energy-aware objectives alongside latency and throughput, particularly for edge and low-power node classes; and conducting a testbed evaluation on production-representative workload traces to validate the simulation-based findings under real scheduling and profiling overhead.

REFERENCES

[1] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley, "An experimental time-sharing system," in Proc. Spring Joint Computer Conference, 1962, pp. 335–344.
[2] A. Cobham, "Priority assignment in waiting line problems," Journal of the Operations Research Society of America, vol. 2, no. 1, pp. 70–76, 1954.
[3] R. Buyya, R. Ranjan, and R. N. Calheiros, "Modeling and simulation of scalable cloud computing environments and the CloudSim toolkit," in Proc. High Performance Computing & Simulation Conf. (HPCS), 2009, pp. 1–11.
[4] L. Kleinrock, Queueing Systems, Volume 1: Theory. New York: Wiley, 1975.
[5] S. K. Garg, S. Versteeg, and R. Buyya, "A framework for ranking of cloud computing services," Future Generation Computer Systems, vol. 29, no. 4, pp. 1012–1023, 2013.
[6] Z. Xiao, W. Song, and Q. Chen, "Dynamic resource allocation using virtual machines for cloud computing environment," IEEE Trans. Parallel and Distributed Systems, vol. 24, no. 6, pp. 1107–1117, 2013.
[7] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," IEEE Trans. Parallel and Distributed Systems, vol. 13, no. 3, pp. 260–274, 2002.
[8] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC), 2011, pp. 1–12.
[9] S. Singh and I. Chana, "QRSF: QoS-aware resource scheduling framework in cloud computing," Journal of Supercomputing, vol. 71, no. 1, pp. 241–292, 2015.
[10] W. Zheng and R. Sakellariou, "Budget-deadline constrained workflow planning for admission control in market-oriented environments," in Proc. Int'l Conf. Cloud and Green Computing, 2012, pp. 105–112.
[11] Y. Ding, X. Qin, L. Liu, and T. Wang, "Energy efficient scheduling of virtual machines in cloud with deadline constraint," Future Generation Computer Systems, vol. 50, pp. 62–74, 2015.

[12] N. Mansouri, R. Ghafari, and B. M. Zade, "Cloud computing simulators: A comprehensive review," Simulation Modelling Practice and Theory, vol. 104, 102144, 2020.
[13] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in Proc. ACM Workshop on Hot Topics in Networks (HotNets), 2016, pp. 50–56.
[14] J. Bi, H. Yuan, and M. Zhou, "Temporal prediction of multi-application consolidated workloads in distributed clouds," IEEE Trans. Automation Science and Engineering, vol. 16, no. 4, pp. 1763–1773, 2019.
[15] P. Mell and T. Grance, "The NIST definition of cloud computing," NIST Special Publication 800-145, National Institute of Standards and Technology, 2011.
[16] L. Wang, G. von Laszewski, A. Younge, X. He, M. Kunze, J. Tao, and C. Fu, "Cloud computing: A perspective study," New Generation Computing, vol. 28, no. 2, pp. 137–146, 2010.
[17] S. Shaw and A. K. Singh, "A survey on scheduling and load balancing techniques in cloud computing environment," in Proc. Int'l Conf. Computer and Communication Technology (ICCCT), 2014, pp. 87–95.
[18] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," IEEE Internet of Things Journal, vol. 3, no. 5, pp. 637–646, 2016.