

Comparing Elitist Genetic Refinement and Reference-Point-Based Multi-Objective Optimization for Deep Q-Network Cloud Task Scheduling

Sara Adnan Mahmood^{1*}, Hussein Najm Abd Ali²

Department of Computer Science, College of Education for Pure Science, University of Wasit, Wasit, Iraq.

E-mail: ^{1*}std.2025205.sar.mahmood@uowasit.edu.iq, ²hnajim@uowasit.edu.iq

Orcid ID: ^{1*}0009-0005-5701-028X, ²0009-0008-9305-6789

Abstract

Cloud task scheduling must balance conflicting objectives: deadline compliance, energy consumption, makespan, and resource utilization. A common design pairs a forecasting module and a reinforcement-learning placement agent with a periodic evolutionary refinement stage, usually a single-objective genetic algorithm that merges the four objectives into one weighted score. This paper studies whether replacing that stage with the reference-point-based multi-objective Non-dominated Sorting Genetic Algorithm III (NSGA-III) changes the scheduling results when forecasting and placement remain unchanged. Both strategies were evaluated on simulated five-machine clusters with workload-prediction signals from real monitoring traces over ten random seeds at 50,000 tasks (NSGA-III also at 100,000). The elitist genetic algorithm produced a mean of 0.495 percent of tasks violating their service-level agreement (SLA), against 0.178 percent for NSGA-III; a paired Wilcoxon signed-rank test confirmed the difference ($p = 0.0039$, $r = 0.854$). Doubling the workload left NSGA-III's violation rate stable (paired $p = 0.426$). NSGA-III needed far more time per run (48.56 versus 9.05 minutes) and slightly more energy (30,582,411 versus 30,034,047 simulation units; $p = 0.0020$, $r = 0.886$). Sensitivity analyses on a 10,000-task protocol showed that the result depends on configuration: ten tasks per batch lay in a narrow working range, the number of reference points mattered little, and the final-selection weights mattered most. The two procedures also differ in search budget, but a genetic algorithm given NSGA-III's budget and operators had a higher mean violation rate than either of them, so extra search does not explain the advantage. With task attributes derived from the Bitbrains traces, the ranking held, but the SLA advantage of NSGA-III shrank from 65 to 24 percent, and its energy penalty disappeared.

Keywords: Cloud Resource Scheduling, Deep Q-Network, Genetic Algorithm, NSGA-III, Multi-Objective Optimization, Statistical Significance Testing.

1. Introduction

Cloud data centers constantly offload a varying stream of incoming tasks to a limited pool of virtual machines. The quality of the offloading affects three outcomes that matter to provider

* Corresponding Author

Journal of Trends in Computer Science and Smart Technology, December 2026, Volume 8, Issue 4, Pages 993-1017

DOI: <https://doi.org/10.36548/jtcsst.2026.4.005>

Received: 19.08.2026, received in revised form: 21.09.2026, accepted: 12.10.2026, published: 24.10.2026

© 2026 Inventive Research Organization. This is an open access article under the Creative Commons Attribution-NonCommercial International (CC BY-NC 4.0) License

and tenant alike: the speed of finishing submitted work, the amount of electricity the facility consumes, and whether the deadlines in service-level agreements (SLAs) are met. Rule-based schedulers such as Round Robin and First-Come-First-Served (FCFS) react only to the current state of the cluster and cannot look ahead to near-future demand, which has led a large body of research to study learning-based and evolutionary alternatives (Arunarani et al. [1]; Bodra and Khairnar [2]). The stakes are more than academic: providers pay directly for every watt a data center draws, tenants hold providers to contractual deadlines that carry financial penalties when missed, and the same cluster typically has to serve both goals at once under demand that keeps shifting. A scheduler that only reacts to the moment leaves real savings and real compliance margin on the table.

1.1 Background and Motivation

The field's convergence on staged design, rather than an end-to-end approach with a single agent, warrants examination. Integrating end-to-end. Folding forecasting, placement, and refinement into a unified policy would necessitate reasoning across a multi-step demand forecast, a sequential placement decision, and a multi-objective trade-off among several objectives, all driven by one reward signal. In practice this combined search space tends to slow learning and makes it hard to tell which part of the behavior is responsible for a given outcome. Splitting the problem into three narrower stages keeps each component accountable to a clearer objective. The present study follows this established design and focuses on how the final refinement stage should reconcile competing objectives.

A typical design integrates three complementary components into a single pipeline. A forecasting module, typically using recurrent or attention-based networks, reads historical monitoring data and predicts near-future resource demand. A deep reinforcement-learning agent, such as the policy-based agent of DeepRM or a Deep Q-Network (DQN), learns by trial and error against the cluster to select a virtual machine for each arriving task (Mao et al. [3]). An evolutionary refinement stage, normally a genetic algorithm, then reviews batches of recent placement decisions and revises them collectively, because a single greedy choice can seldom balance conflicting goals such as saving energy while still meeting deadlines.

The refinement stage of this pipeline, as described in published versions, employs a genetic algorithm (GA) that folds every scheduling objectives into a single fitness value through a weighted sum. These weights are set, the trade-off between energy savings and deadline risk is fixed in advance and stays frozen even if the real cost of a missed deadline or a wasted watt shifts between workloads. Several recent studies have proposed sharpening the evolutionary component itself, among them an elitism-based genetic algorithm for task scheduling in Infrastructure-as-a-Service (IaaS) clouds (Osama et al. [4]) and an adaptive elitism design paired with conditional parameter tuning, reported to shorten makespan against Min-Min, Max-Min and two genetic-algorithm baselines on Amazon EC2 instance types (Osama et al. [5]). Yet how dependable elitism is on its own is not settled. In preliminary runs by the authors, an apparent 49% SLA gain for elitist selection at three seeds came from comparing runs whose environments differed (the deadline rule and the forecast array); under identical conditions elitism gave no gain and, at ten seeds, a higher violation rate (Section 4.1). That experience raises the question pursued here: could this unreliability come from protecting strong solutions between generations, or from the overly simple rule used to decide which solution is strong in the first place? This study addresses that instability by running both the

genetic algorithm and NSGA-III under the same ten-seed protocol and reporting effect sizes alongside significance tests rather than single-run point estimates.

The Non-dominated Sorting Genetic Algorithm III (NSGA-III) offers a different answer to that question and to the frozen-tradeoff problem described above. Rather than combining the objectives into a single score, it handles them individually and selects survivors according to their position on the Pareto front. To keep the surviving solutions diverse, it uses a set of evenly distributed reference points (Deb and Jain [6]). The algorithm was built specifically for problems with three or more simultaneous objectives, a range in which its predecessor, the Non-dominated Sorting Genetic Algorithm II (NSGA-II), is documented to struggle (Wietheger and Doerr [7]). NSGA-II was deliberately left out of the experimental comparison for this reason. Its crowding-distance mechanism ranks solutions within a front by local density rather than by distance to a predefined reference direction, and is known to lose discriminating power once three or more objectives are optimized; including it would have measured a known weakness rather than added a competitive alternative. NSGA-III has already been applied on its own to cloud task scheduling (Latreche et al. [8]) and paired with reinforcement learning for related scheduling problems in manufacturing settings (Tang et al. [9]), but no published study known to the authors has tested it as a periodic refinement stage on top of an independently trained DQN placement agent within the three-stage pipeline described above.

This paper reports a controlled comparison in which the forecasting and placement stages of an existing hybrid pipeline were held fixed while the refinement stage was swapped between an elitist genetic algorithm and NSGA-III. Both variants were tested using the same statistical protocol, ten independent random seeds each, in keeping with the earlier elitism finding, and outcomes were compared using non-parametric significance testing suited to small, non-normally distributed samples (Derrac et al. [10]; Carrasco et al. [11]). Section 2 surveys related work on DQN-based placement, evolutionary refinement, and Transformer-based forecasting. Section 3 describes the shared architecture and the two refinement variants in detail. Section 4 reports results for each refinement strategy on its own, their direct statistical comparison, a stability check across workload sizes, and sensitivity analyses (Sections 4.6 to 4.11), followed by a discussion of these findings in relation to prior work and practical trade-offs (Sections 4.12 to 4.15). Section 5 concludes.

2. Related Work

2.1 DQN-Based Cloud Scheduling

Before surveying how prior work has adapted it to cloud scheduling, Figure 1 shows how the Deep Q-Network is used for placement in this study. As each task arrives, the network reads the current cluster state, the Stage-1 forecast, and the task's demand and proposes a virtual machine (VM) for it; once ten tasks have accumulated, the refinement stage revises their assignments, the tasks are placed, and the simulator advances one time step. The reward-driven weight updates happen only during training, which is described in Section 3.

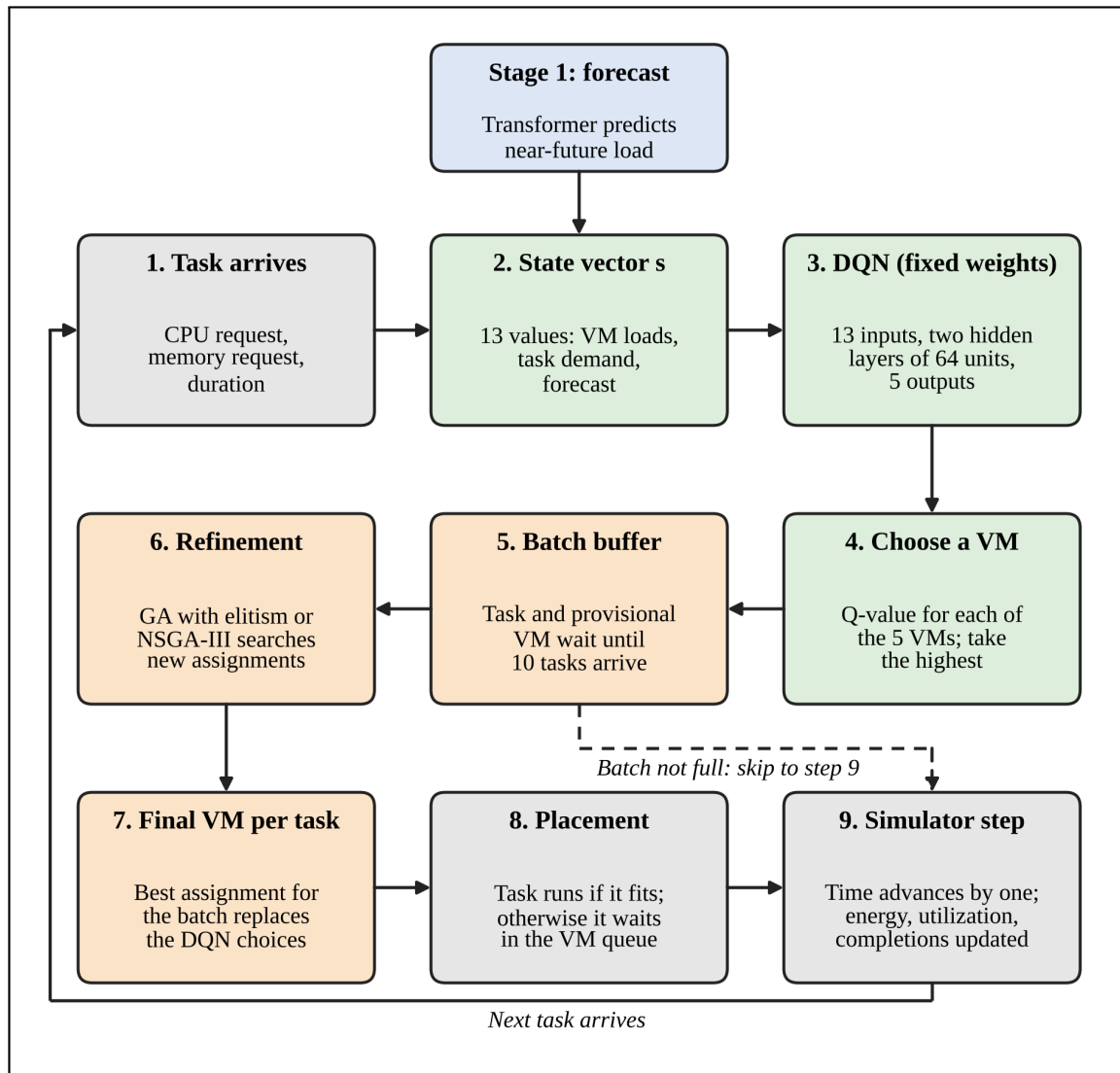


Figure 1. How tasks are placed in this study. The DQN picks a virtual machine for every task that arrives; after ten tasks, the refinement stage revises the picks for the whole batch, and only then are the tasks placed

The notion that task placement can be learned from experience, rather than from hand-written rules, goes back to DeepRM, one of the first systems to demonstrate that a trained policy can perform about as well as manually tuned heuristics on multi-resource packing problems (Mao et al. [3]). Building on this foundation, a Deep Q-LSTM design paired recurrent memory, known as Long Short-Term Memory (LSTM), with DQN to capture temporal dependencies in workload traces, reporting gains in service time and virtual-machine utilization over comparison methods (Xing [12]). A lightweight DQN scheduler with a single-layer LSTM state encoder was reported to train on CPU alone and to be validated on workloads from one hundred to one hundred thousand tasks (Naaz et al. [13]). Elsewhere, an improved double DQN targeted makespan minimization in edge computing by separating the calculation of target values from action selection, reducing the overestimation bias known to affect standard DQN (Zeng et al. [14]). A maintenance-aware adaptive double DQN has also been proposed for cloud-manufacturing scheduling, using a two-layer architecture that first selects optimization weights and then applies composite scheduling rules under a multi-dimensional reward (Zhang et al. [15]). Reinforcement learning combined with cost- and

energy-aware objectives has likewise been reported for dynamic multi-objective scheduling: a Deep Q-Network scheduler of this kind was reported to cut energy consumption by up to 27 percent and improve cost efficiency by up to 18 percent against heuristic and metaheuristic methods while meeting deadline constraints (Yu et al. [16]).

Combining reinforcement learning with imitation-based guidance has also been shown to speed up learning convergence and shorten weighted job turnaround time (Guo et al. [17]).

2.2 Evolutionary and Multi-Objective Refinement

Genetic algorithms remain a standard tool for revisiting scheduling decisions in batches rather than one at a time, since batch-level search can weigh several criteria together in a way a single greedy step cannot. An elitism-based genetic algorithm for task scheduling in IaaS clouds was reported to beat conventional scheduling methods on Amazon EC2 in how quickly tasks finished, how well resources were used, and how fast the search converged (Osama et al. [4]). A related adaptive elitism design combined with conditional parameter tuning reported faster convergence and shorter makespan than Min-Min, Max-Min, a standard genetic algorithm, and an adaptive-incremental genetic algorithm on Amazon EC2 instance types, in an experiment with ten tasks (Osama et al. [5]). On the multi-objective side, NSGA-III was applied to cloud task scheduling to jointly minimize runtime, power draw, and cost, with reported results showing it ahead of NSGA-II (Latreche et al. [8]). A formal runtime analysis later supported this comparison: NSGA-III computes the full Pareto front of a three-objective benchmark in expected $O(n \log n)$ iterations when given enough reference points, whereas NSGA-II had been shown to miss a constant fraction of it (Wietheger and Doerr [7]). A double-Q-learning-guided NSGA-III has also been proposed for flexible job-shop dynamic scheduling, using reinforcement learning to adaptively tune the evolutionary algorithm's own parameters rather than fixing them in advance (Tang et al. [9]). NSGA-II itself remains a strong and actively used baseline in this space. A 2026 federated multi-cloud scheduler combining NSGA-II with Q-learning reported makespan reductions of up to eighteen to thirty-two percent and utilization gains of ten to twenty-two percent over existing baselines (including FCFS, Max-Min, and a multi-objective particle-swarm variant) across synthetic, Google Cloud, and IoT workloads (Ghaban and Alatawi [18]). The choice between NSGA-II and NSGA-III in a given pipeline is therefore a genuine trade-off between search reliability on many objectives and per-generation cost, not a foregone conclusion in favor of NSGA-III. More broadly, multi-objective genetic algorithms have been used for dynamic cloud resource optimization aimed jointly at energy efficiency, load balancing, and cost (Guna et al. [19]), and an improved NSGA-II variant has been proposed for emergent resource demands, aiming at better solution quality, more uniform solution distribution and faster solving (Chen et al. [20]). More recently, a modified NSGA-III has been integrated into the critic of a Double Dueling DQN, where it was used to balance cost and performance objectives during the actor-critic training loop itself (Felsy and Sajan [21]). The pipeline studied here, however, keeps the placement agent's training fully decoupled from the refinement mechanism, with NSGA-III only applied afterward as a periodic, batch-level correction rather than as part of the learning process.

2.3 Transformer-Based Workload Forecasting

Attention-based forecasting has been increasingly used for cloud workloads; it can weight distant time steps directly, instead of passing information through the step-by-step

chain of a recurrent network. An early Wasserstein-adversarial Transformer was reported to achieve inference about five times faster with up to 5.1 percent higher accuracy compared to a state-of-the-art LSTM baseline, and was further shown to improve auto-scaling on a real cloud platform (Arbat et al. [22]). A transformer-based performance predictor is used with a reinforcement-learning controller to proactively assign resources for cloud-native microservices by adjusting the CPU, memory, and disk I/O based on the predicted load on the critical execution path (Chen et al. [23]). More recent work has combined Transformers with differential-evolution tuning to handle non-stationary workload patterns (Zhang et al. [24]), integrated wavelet transforms with contrastive learning to improve cloud load prediction accuracy (Zhang et al. [25]), and introduced a feature-enhanced, similarity-aware Transformer that groups containers with similar workload patterns for large-scale prediction at lower overhead (Ding et al. [26]). Transformer-based predictors have been reported in these studies to outperform recurrent alternatives in accuracy, inference speed, or both, which motivates their continued use as the forecasting component in hybrid scheduling pipelines such as the one considered here.

Results from the above studies indicate that DQN-based placement, evolutionary refinement, and Transformer-based forecasting have each been studied extensively, and combinations of two of the three are gaining traction. What has not been investigated, and is the focus of this paper, is the effect of upgrading the refinement stage from a single-objective genetic algorithm to NSGA-III when it is the only component changed in an existing Transformer-plus-DQN pipeline. It is worth noting that the forecasting module used in the present pipeline is univariate: it consumes a single, aggregated cluster-load signal drawn from the historical monitoring window and returns one scalar estimate, rather than modeling CPU, memory, and other resource metrics as separate, jointly predicted series.

3. Proposed Work

The scheduling problem is approached as a sequential decision-making process within a cluster of five VMs. The CPU and memory capacity of each VM are uniformly distributed between fifty and one hundred units. Tasks arrive one at a time asking for between 5 and 25 units of CPU and memory, and run for between 3 and 14 simulated time steps. Each task has a deadline equal to its arrival time plus 4.5 times its duration. If the task is completed after this deadline, the SLA is violated. The instantaneous power draw of each machine is modeled as a linear function as shown in Equation (1). The power varies linearly from an idle minimum to a fully loaded maximum, depending on the CPU utilization.

$$P = P_{idle} + (P_{max} - P_{idle}) \times u \quad (1)$$

The baseline power, P_{idle} , is 50 plus three-tenths of the machine's CPU capacity, and the maximum power, P_{max} , is an additional 150 units on top of this baseline; all energy values reported in this paper are the sum of this per-machine, per-step power draw across the entire simulated run. P_{idle} , P_{max} , and the total power draw are all expressed in abstract simulated power units rather than physical watts, consistent with the simulation-based nature of the cluster model; this same unit convention applies to every energy figure reported elsewhere in the paper. In simple terms, a machine consumes baseline power merely by being switched on, and this grows steadily to a fixed ceiling as its CPU fills up, which is how physical servers behave rather than toggling between two fixed power states. Two further quantities are used in the results:

makespan is the number of simulated time steps from the start of a run until all tasks within it have finished, and the SLA violation rate is the percentage of finished tasks that finished after their deadline. The results show the CPU and memory consumption separately, averaged first over the five machines in the cluster, and then over each simulated step. The utilization term in the fitness and reward equations below is used to monitor CPU utilization in particular, since CPU capacity is the resource most directly contended by placement decisions. The complete pipeline consists of three stages, shown in Figure 2.

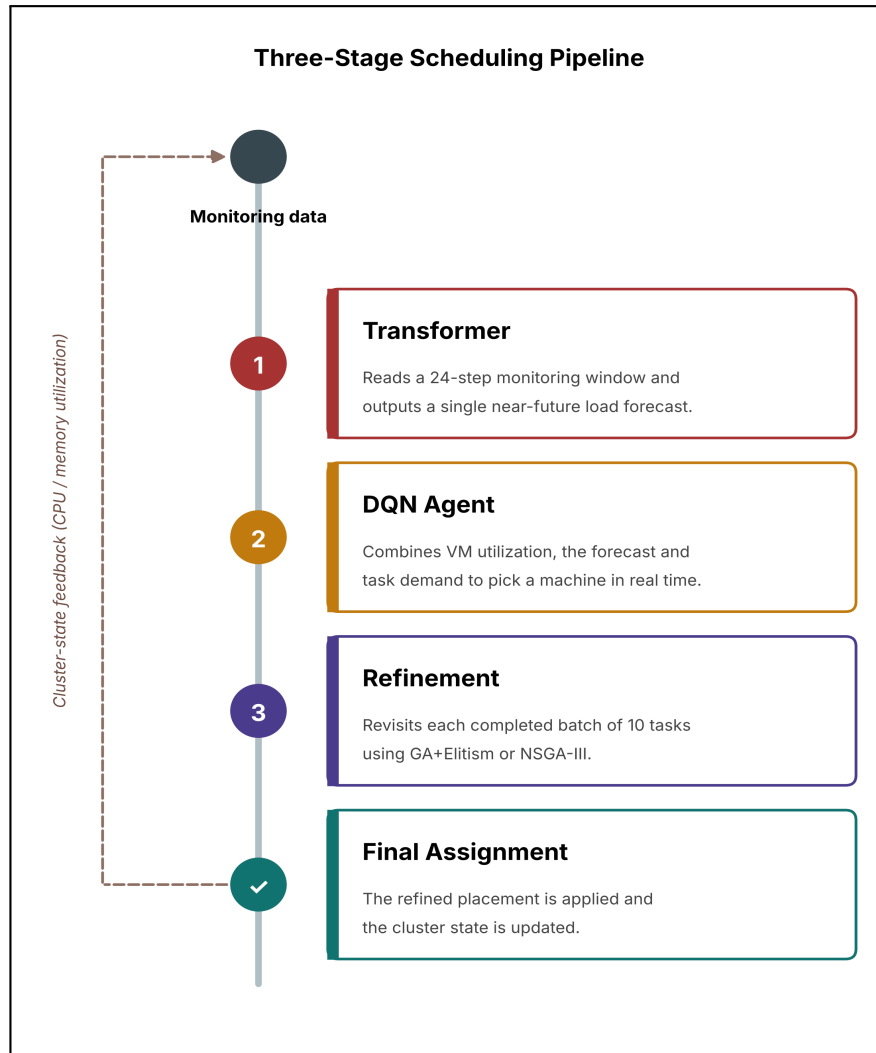


Figure 2. Overview of the three-stage scheduling pipeline

3.1 Stage 1: Workload Forecasting

A Transformer-based predictor reads a rolling window of 24 historical monitoring intervals and outputs a single scalar estimate of near-future cluster load. The model stacks two multi-head self-attention blocks, each with two attention heads and a key dimension of 32, followed by a feed-forward sub-layer of 64 rectified linear units, residual connections, and layer normalization. This architecture is shown in Figure 3. The forecast is appended to the current utilization of each machine and the resource demand of the arriving task, forming the observation passed to the placement agent in Stage 2.

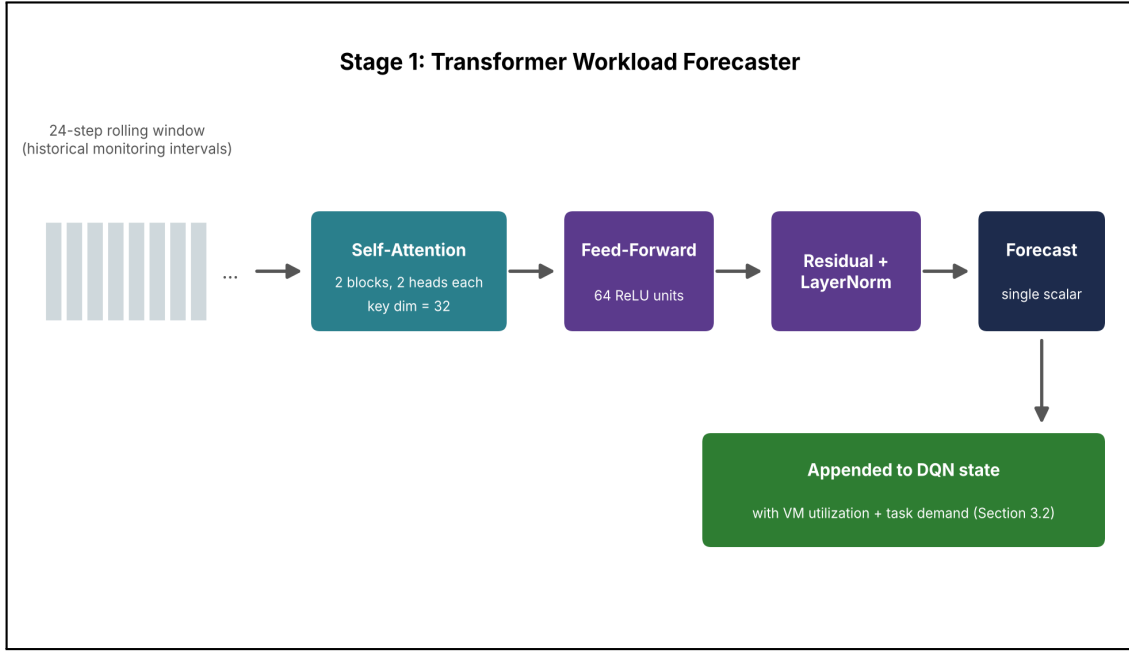


Figure 3. Architecture of the transformer-based workload forecaster used in Stage 1

3.2 Stage 2: Real-Time Placement

A Deep Q-Network (DQN) picks a machine for the arriving task using a feed-forward network with two hidden layers of sixty-four rectified linear units. The agent takes in the current utilization of each machine, the Stage-1 forecast, and the resource demand of the arriving task and outputs an estimated value for each of the five candidate machines; the machine with the highest value is chosen. The values are learned using the standard Q-learning update rule from Equation (2), which trains the network to minimize the difference between its current estimate and a target built from the immediate reward plus a discounted estimate of the best action available in the next state. Here s is the state, which includes the current utilization of the cluster, the current Stage-1 forecast, and the resource demand of the arriving task, and a is the choice among the five candidate machines. In plain terms, the agent's estimate of how good a placement was is nudged toward what actually happened plus its own best estimate of what follows, so its estimates become more accurate with every placement.

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \quad (2)$$

The learning rate α was fixed at 0.001 with the Adam optimizer, and the discount factor γ was set to 0.95, giving the agent a moderate preference for reward gathered over the next several placements rather than only the immediate one. Exploration followed an epsilon-greedy schedule, decaying from 1.0 at the start of training to a floor of 0.05 at a rate of 0.97 per episode. Past transitions were stored in a replay buffer of 10,000 transitions, sampled in mini-batches of 64, and the target network was synchronized with the main network every ten episodes to keep the training target from shifting with every update. The agent was trained for 100 episodes before the evaluation reported in Section 4.

The placement decisions of the agent during training were governed by the reward function in Equation (3) composed of five weighted terms: normalized step energy (weight 0.45), penalty for actions that cannot be admitted immediately (0.25), current queue length (0.15), penalty for overrun of makespan (0.10), and mean utilization (0.05, rewarded rather

than penalized). The energy is weighted the most since the main goal of placement phase is to reduce the power draw. These weight values were manually set to reflect the project priorities, with energy consumption receiving the largest share, followed by blocked admissions, and were kept constant across all experiments reported here so that the genetic-algorithm and NSGA-III refinement variants could be compared under the same reward signal. The agent is trained to view energy, blocked admissions, queueing, and overrun time as costs worth avoiding and high utilization as a bonus worth chasing, but only as a minor priority behind saving power, since its weight is only 0.05, and it is the only term that is added rather than subtracted. At evaluation time, the weights of the network are fixed, and each placement decision is read directly from the trained policy by selecting the machine with the highest estimated value, as in Equation (2). Equation (3) is not used during evaluation. In Equation (3), E, S, Q, K, and U are the five terms in the order listed above; S is the blocked-admission penalty and is distinct from the SLA term S in Equation (4).

$$r = -0.45E - 0.25S - 0.15Q - 0.10K + 0.05U \quad (3)$$

3.3 Stage 3: Periodic Refinement — Two Compared Variants

Every ten successive placement decisions are grouped into a batch and revisited before being locked in. This is the only stage that differs between the two configurations compared in this study.

3.3.1 Elitist Genetic Algorithm (Baseline Configuration)

The population of twenty candidate solutions, each consisting of a vector of ten machine indices, evolves over twenty generations. Each vector position is assigned to one of the ten tasks in the current batch, with the value at that position indicating the virtual machine to which the candidate would send that task; the initial population is seeded with the DQN's own assignment plus a set of lightly mutated variants, so the search begins close to what the placement agent suggested previously rather than from an arbitrary point. In each generation the twenty candidates are scored and ranked, the best ten are kept as parents, and nine children are produced by randomly choosing two of these parents, splitting both of their vectors at a single randomly chosen point and swapping their tails (single-point crossover); no mutation is applied after the initial population. Fitness combines the same four objectives into a single weighted score, as shown in Equation (4), with SLA compliance given the largest nominal weight (0.40) to counteract the energy-first reward driving placement (see the note on the evaluator below). The best candidate of each generation is also copied unchanged into the next one (an elitism step), which keeps the population at twenty, and the best assignment found over all generations, including the DQN's own, is the one returned.

$$f = -(0.40S_n + 0.30E_n + 0.20K_n - 0.10U_n) \quad (4)$$

Before entering Equation (4), the four objectives of each candidate are rescaled by fixed divisors: S is the SLA violation rate in percent divided by 20, E is total energy divided by 30,000, K is the batch's completion time divided by 100, and U is mean CPU utilization in percent divided by 60, the same divisors used throughout candidate evaluation in both refinement variants. Utilization enters with a negative sign inside the bracket, so higher utilization lowers the weighted cost; f is the negative of that cost, and the candidates with the highest f (the lowest weighted cost) are kept. This fitness function rewards batch assignments

that keep energy draw and completion time low while pushing utilization up, with SLA violations entering the score as described below. The same four-way balance Equation (3) pursues one placement at a time, now applied to a whole batch at once.

One property of this evaluator should be noted. Each candidate batch is scored on an otherwise idle cluster, and a task's deadline inside the evaluator is measured from the step at which it begins executing rather than from its arrival. Waiting time inside the batch therefore never counts as lateness: in our checks, including a worst-case assignment that places every task of a batch on a single machine, the SLA term was zero for every candidate, so the 0.40 weight on it did not influence the ranking of candidates. Refinement acts on SLA only indirectly, through energy, makespan, and utilization, whereas the SLA rates reported in Section 4 are measured by the full simulator, where deadlines run from arrival.

3.3.2 NSGA-III (Proposed Alternative)

Rather than merging the four objectives into one score, NSGA-III keeps them apart and searches for a Pareto front, a set of solutions where none is worse than another across every objective at once (Figure 4). The pool of forty (twenty parents and twenty offspring produced by single-point crossover and a small chance of mutation) is sorted here by non-domination. A candidate dominates another if it matches or beats it on all four objectives and beats it outright on at least one. The candidates that no other candidate dominates form the first front; removing them and repeating the comparison on what remains produces a second front, and so on until every candidate has been placed. Population slots are then filled front by front, starting with the strongest, until adding the next whole front would overshoot the limit of twenty. Eighty-four evenly spread reference points, built using the Das-and-Dennis approach adopted in the original NSGA-III formulation (Deb and Jain [6]), decide which candidates from that final, over-sized front make the cut. Each of the four objective values is first rescaled so that the best and worst performers on that objective sit at the same relative extremes. Each candidate is then matched to whichever of the eighty-four reference points its rescaled objective values sit closest to. This spreads the surviving solutions across the whole range of trade-offs between energy, SLA compliance, makespan, and utilization instead of collecting them around the best trade-off on a single objective. When the last front needs to be trimmed to fill the remaining slots, candidates that share their reference point with fewer other members of that last front are preferred to candidates whose reference point is already crowded. The population size (twenty), number of generations (twenty), batch size, and fitness evaluator are the same as in the genetic algorithm, but the search budget is not: NSGA-III evaluates forty candidates per generation and 820 per batch against 401 for the genetic algorithm and uses mutation, which the genetic algorithm does not. The comparison is therefore between the two refinement procedures as implemented, not between selection rules under an identical budget; Section 4.11 adds a control with an identical budget. Because a single assignment must be deployed, the candidate returned after the 20th generation is selected from the last population with the same weighted-sum formula as Equation (4) (see Algorithm A1 in Annexure A, Step 15). This scalarization step is performed only once, after the Pareto-based search has run its full course, solely to select a single representative solution for deployment and reporting; it does not affect the survival of candidates during the twenty generations of search itself, where selection is driven exclusively by non-dominated sorting and reference-point association. The technique for order of preference by similarity to ideal solution (TOPSIS) could have been used for the knee-point selection or ranking equally well for this last step, but the weighted-sum rule was

retained because it is the criterion that already drives the search of the genetic algorithm, so using it on both final populations restricts the comparison to the search mechanism.

This design choice has a consequence that should be stated plainly. The Pareto-based component of NSGA-III determines which candidates survive during the search, whereas the assignment finally deployed is chosen by a preference-dependent weighted-sum rule. The pipeline is therefore a two-stage procedure, Pareto-based search followed by a posteriori scalarization, and not a method that hands the operator a Pareto set or removes the need for preference weights; the weights in Equation (4) remain an input that is held fixed across all experiments. The comparison reported here should accordingly be read as one between two search mechanisms under an identical final-selection rule, and the observed differences are attributable to that combined pipeline, not to Pareto optimization in isolation. Figure 4 contrasts the two selection strategies on an illustrative set of candidate solutions.

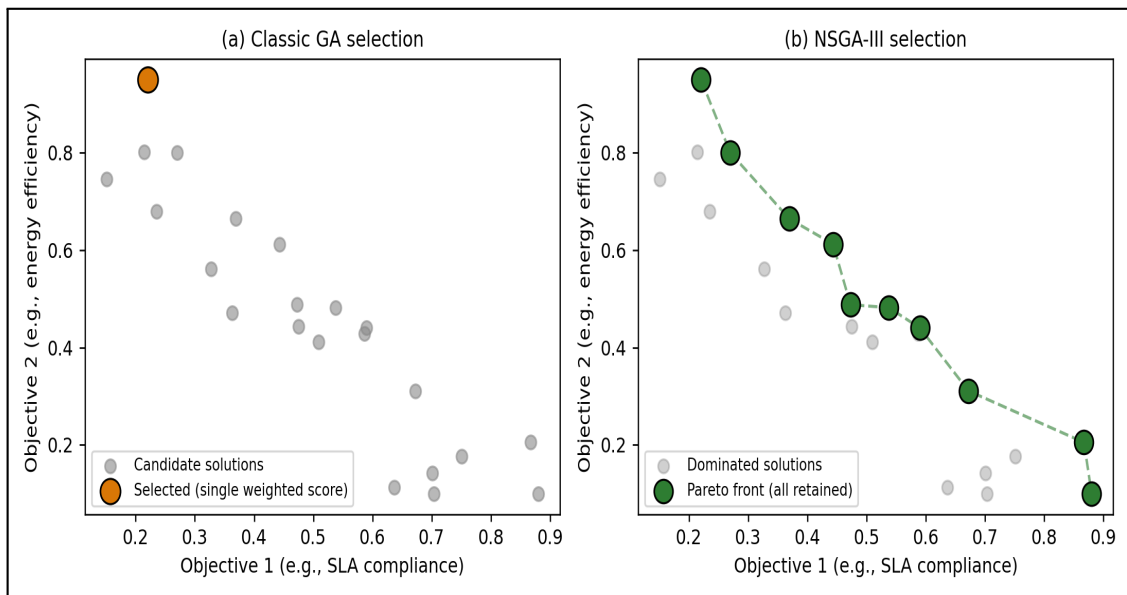


Figure 4. Selection strategy comparison: (a) GA's single-score approach; (b) NSGA-III's Pareto front. Both axes are normalized scores without units, and the points are illustrative: in the implemented pipeline, both procedures finally deploy the single candidate with the best weighted score of equation (4)

3.4 Overall Scheduling Flow

Figure A1 (Annexure A) describes the decision flow for each incoming task, from its arrival until the cluster state is updated after placement, and shows where the two refinement variants diverge.

Algorithm A1 (Annexure A) gives the refinement procedure in pseudocode for both variants, which diverge in Steps 3 to 13.

3.5 Workload and Data

Two publicly available workload traces were used; the DQN placement agent was trained only on the trace of the Google Borg cluster (system described in Verma et al. [27]). The Bitbrains GWA-T-12 trace (Shen et al. [28]) on monitored virtual-machine behavior for financial sector customers was sequestered entirely to generate the workload forecasting signal used at evaluation time and was not touched during training. The reported results therefore represent a genuine transfer to monitoring conditions not seen by the trained components. Monitoring

windows from both traces were fed directly to the Transformer, whereas the per-task attributes in the simulated environment (CPU request, memory request, and duration) were drawn from uniform synthetic distributions rather than from per-task fields in the traces (Section 4.10 repeats a reduced comparison on attributes derived from the Bitbrains traces). This is stated plainly: the comparison in Section 4 concerns the effect of the refinement mechanism on a shared, controlled workload, not a claim of full statistical-structure generalization across independently collected traces. The same generated workload instances were used for each of the seeds for both refinement configurations, so any differences in results arise from the refinement procedure as implemented (Section 3.3.2), not from differences in the task stream.

4. Results and Discussion

All experiments were carried out using a custom discrete-event simulator implemented in Python, featuring five virtual machines and a workload of 50,000 tasks, unless stated otherwise. Every configuration was repeated for ten independent random seeds (42 to 51) and the results shown are averaged over these repetitions.

4.1 Elitist Genetic Algorithm — Results

Table 1. GA+Elitism performance across ten independent seeds (50,000 tasks)

Seed	SLA Violation (%)	Energy (Simulation Units)
42	0.226	29,653,250
43	0.698	30,696,859
44	0.288	30,213,776
45	0.150	29,200,071
46	0.680	30,467,623
47	1.000	30,848,238
48	0.190	29,574,902
49	0.540	29,868,281
50	0.360	28,975,653
51	0.820	30,841,816
Mean	0.495 ± 0.295	$30,034,047 \pm 681,014$

Each row in Table 1 is equivalent to a single independent trial. A different random seed produces a different sequence of arriving tasks, so the SLA and energy numbers in a single row relate to how the genetic algorithm handled that particular sequence of workloads, not ten repetitions of the same problem. As can be seen in Table 1, the elitist genetic algorithm achieved a mean SLA violation rate of 0.495 percent with a wide variation across the seeds, ranging from 0.150 percent at seed 45 to 1.000 percent at seed 47, showing a real sensitivity to the random task stream that a given trial draws. The energy column is over a narrower band, 28,975,653 at seed 50 to 30,848,238 at seed 47, about six percent spread around the mean. Table 1 reports the original run; a repeat of the same ten seeds reproduced five rows exactly and differed in the other five by at most 0.012 percentage points of SLA violations (mean 0.494 percent), and the paired comparison with NSGA-III gives the same result with either set of values ($p = 0.0039$).

As a check on the elitism step, the genetic algorithm was also run without it, in exactly the same environment (same placement agent, forecast array, deadline rule, and ten seeds). The variant without elitism reached a mean SLA violation rate of 0.424 percent against 0.494 percent with elitism (lower in nine of ten seeds; exact Wilcoxon $p = 0.004$), with almost identical mean

energy (30,064,514 versus 30,041,162 simulation units; 0.08 percent lower with elitism, $p = 0.020$). Elitism, therefore, did not help in this setting. NSGA-III (0.178 percent) was also better than the variant without elitism (nine of ten seeds; exact $p = 0.004$), so the comparison in Section 4.3 does not depend on which genetic-algorithm variant serves as the baseline.

4.2 NSGA-III — Results (50,000 Tasks)

Table 2. NSGA-III performance across ten independent seeds (50,000 tasks)

Seed	SLA Violation (%)	Energy (Simulation Units)
42	0.272	30,779,604
43	0.314	31,614,539
44	0.064	30,914,845
45	0.012	29,375,466
46	0.140	31,148,964
47	0.536	31,093,538
48	0.024	29,727,992
49	0.094	29,971,061
50	0.122	30,076,740
51	0.204	31,121,355
Mean	0.178 ± 0.161	$30,582,411 \pm 738,134$

Table 2 shows a markedly lower mean SLA violation rate of 0.178 percent under NSGA-III, with every seed except 43 and 47 landing under 0.3 percent and four seeds (44, 45, 48, 49) dropping below 0.1 percent. Seed 47 remained the most challenging trial under both configurations (0.536 percent here against 1.000 percent under the genetic algorithm), hinting that this particular random task stream is intrinsically harder for the shared forecasting and placement stages, regardless of which refinement mechanism follows them. Energy consumption under NSGA-III ranges from 29,375,466 at seed 45, the same seed that also posted the lowest SLA violation rate, up to 31,614,539 at seed 43. Unlike the genetic algorithm's results, the seed with the single worst SLA figure here (47) is not also the seed with the highest energy draw, which is an early sign that NSGA-III's energy and SLA outcomes are not as tightly coupled to each other as the genetic algorithm.

4.3 Direct Comparison and Statistical Significance

Figure A3 (Annexure A) summarizes the mean SLA violation rate and mean energy consumption for both refinement variants across the ten seeds.

Table 3. Wilcoxon signed-rank test results comparing the two refinement configurations

Comparison	p-value	Effect Size (r)	Interpretation
SLA: NSGA-III vs. GA+Elitism	0.0039	0.854	Statistically significant; large effect
Energy: NSGA-III vs. GA+Elitism	0.0020	0.886	Statistically significant; large effect

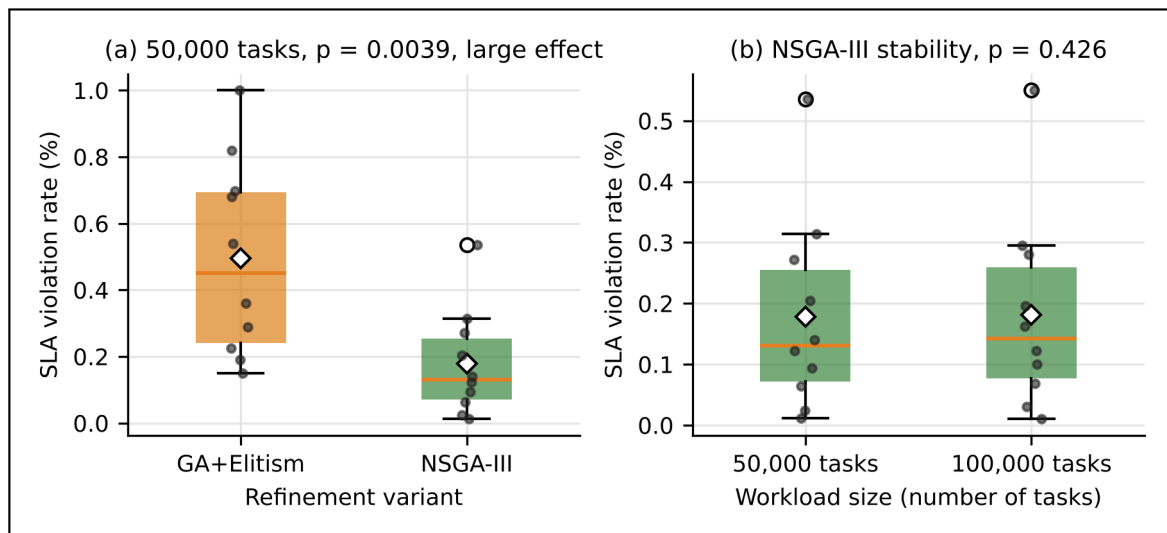


Figure 5. (a) SLA violation rate by refinement variant at 50,000 tasks. (b) SLA violation rate of NSGA-III at 50,000 and 100,000 tasks. Each dot is one seed, the diamond is the mean, the orange line is the median, and the open circle is an outlier. The p-values are the exact paired Wilcoxon signed-rank values

The outcome of that test for each metric is shown in Table 3. Because the same ten random seeds were used to generate the workload for both refinement configurations, each seed yields a matched pair of observations, one from GA+Elitism and one from NSGA-III, rather than two independent samples; the Wilcoxon signed-rank test was therefore used, since it is the non-parametric test for paired comparisons and makes no assumption of normality, which a ten-seed sample would not support. The effect size r rescales the comparison onto a 0-to-1 scale, where values above roughly 0.5 are conventionally read as large. This result suggests that the SLA improvement for NSGA-III is unlikely to be due to chance ($p = 0.0039$, well below the usual 0.05 cutoff), and the large effect size ($r = 0.854$) indicates that the difference is not only statistically significant but also practically meaningful. The energy difference reaches the same conclusion: at $p = 0.0020$, it also falls below the 0.05 threshold, with a comparably large effect size ($r = 0.886$), so the small increase in mean energy consumption under NSGA-III is unlikely to be ordinary seed-to-seed variation either, even though it remains modest in absolute terms.

To make the paired structure explicit, Table 3 is computed on the ten per-seed differences between GA+Elitism and NSGA-III, not on two pooled samples. NSGA-III produced a lower SLA violation rate than GA+Elitism in nine of the ten seeds (seed 42 being the exception), with a median per-seed reduction of 0.31 percentage points, and a higher total energy consumption in all ten seeds, with a median increase of 0.48 million simulation units. Because there are only ten pairs, the p-values are exact two-sided values obtained from the full distribution of signed ranks rather than asymptotic approximations; with ten pairs the smallest attainable value is 0.00195, which the energy comparison reaches because all ten differences share the same sign. The effect size r is Z divided by the square root of the number of pairs, with Z taken from the normal approximation, and is reported alongside the exact p-value. The two primary comparisons in Table 3 remain significant after a Bonferroni correction for two tests (threshold 0.025); the many secondary tests in Sections 4.6 to 4.11 are descriptive and were not corrected for multiple comparisons.

Figure 5 shows the seed-level distribution of SLA violation rates for both refinement variants.

4.4 Stability Across Workload Scale

Table 4. NSGA-III performance at 50,000 versus 100,000 tasks (mean of ten seeds)

Workload Size	Mean SLA (%)	Mean Energy	p-value vs. 50,000 (paired Wilcoxon)
50,000 tasks	0.178 ± 0.161	30,582,411 ± 738,134	—
100,000 tasks	0.181 ± 0.161	61,150,795 ± 1,482,742	0.426 (SLA)

Although the raw energy totals in Table 4 roughly double with the task count, which is expected, the meaningful comparison is energy per task, shown in Figure A2 (Annexure A). Table 4 and Figure A2 show that when the workload is doubled from 50,000 to 100,000 tasks, there is no statistically detectable change in SLA performance ($p = 0.426$, paired Wilcoxon signed-rank test on the ten seed-matched values), and energy consumption per task remains within a fraction of one unit between scales. This indicates that NSGA-III's SLA performance is stable across the two workload sizes tested; because the genetic algorithm was not run at 100,000 tasks, the comparison between the two refinement variants is established at 50,000 tasks only.

4.5 Comparison Against Traditional Scheduling Algorithms

The comparison in Sections 4.1 to 4.4 is only within the proposed pipeline. A separate question is how either refinement variant compares with scheduling rules that have no learning component. Round Robin and FCFS, the two rule-based schedulers introduced in Section 1 (Arunarani et al. [1]; Bodra and Khairnar [2]), and a random-assignment rule were run on the same 50,000-task workloads, with the same ten seeds (42 to 51) and the same deadline rule (arrival plus 4.5 times the duration) as the pipeline. Round Robin cycles through the five virtual machines in a fixed order regardless of load; FCFS places each arriving task on the first machine with room for it, or on the least-loaded machine if none has room. These rules place each task immediately on arrival, whereas the pipeline assigns tasks in batches of ten.

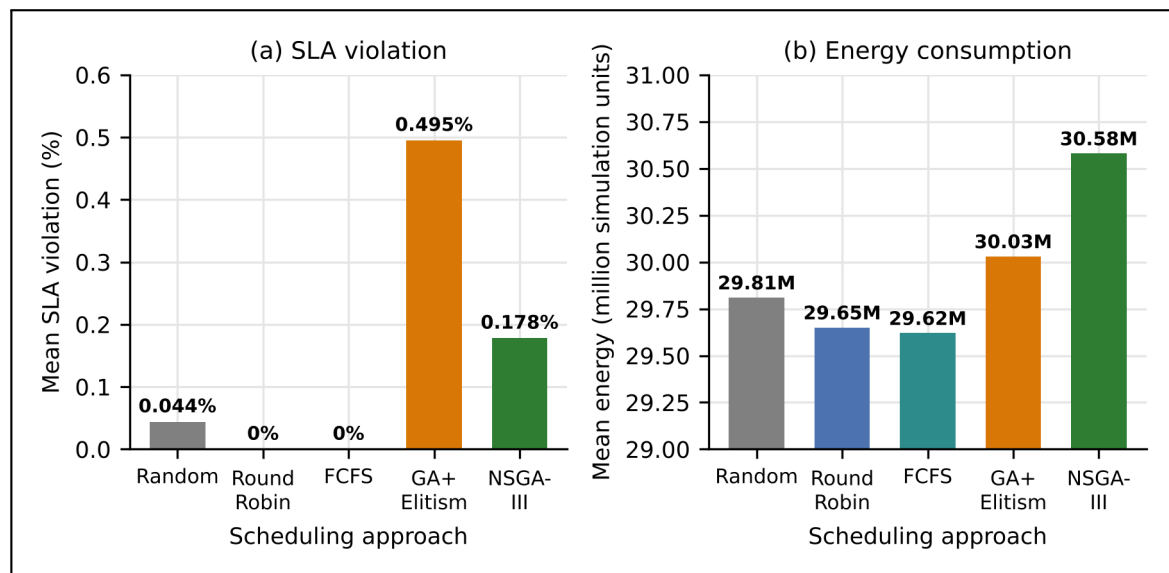


Figure 6. Mean SLA violation rate (a) and mean energy consumption (b) of the five scheduling approaches (50,000 tasks, ten seeds). GA+Elitism and NSGA-III are the two refinement variants studied here. The energy axis in panel (b) does not start at zero

Table 5. Mean SLA violation and energy consumption for two rule-based schedulers, a random-assignment control, and the two refinement variants examined in this paper (50,000 tasks, ten seeds)

Algorithm	Mean SLA Violation (%)	Mean Energy (Simulation Units)
Random assignment	0.044	29,813,781
Round Robin	0.000	29,650,339
First-Come-First-Served	0.000	29,619,234
GA+Elitism (this study)	0.495	30,034,047
NSGA-III (this study)	0.178	30,582,411

Table 5 and Figure 6 show that neither refinement variant outperformed the simple rules. Round Robin and FCFS recorded no SLA violations in any of the ten seeds, and the random rule averaged 0.044 percent, whereas GA+Elitism and NSGA-III had higher violation rates than all three in every seed (exact Wilcoxon $p = 0.002$ for each comparison). The rules also used less energy: the means of Round Robin and FCFS were 1.3 to 1.4 percent below GA+Elitism and 3.0 to 3.2 percent below NSGA-III, and that of the random rule was 0.7 and 2.5 percent below (means only; per-seed energy values were not tested). In this synthetic setting the pipeline therefore did not improve on rules that place each task at once, and Sections 4.1 to 4.4 show only which refinement procedure does better within the pipeline. Part of the SLA gap is the cost of batching: for the agent alone, placing tasks in batches of ten raised violations from 0.02 to 1.77 percent in the 10,000-task protocol of Section 4.6. We did not investigate why the pipeline also used more energy.

4.6 Sensitivity to the Refinement Batch Size

The refinement stage was applied to batches of ten tasks in Sections 4.1 to 4.5, a value fixed in advance rather than tuned. To examine how much the findings depend on it, NSGA-III was rerun with batch sizes of 5, 10, 20, and 50 tasks, using the same ten seeds, the same trained placement agent, and every other setting unchanged. Because each setting requires ten complete runs, this study used a reduced protocol of 10,000 tasks per run instead of 50,000. As a control, the placement agent alone, with no refinement, was run under the same batching scheme and also with a batch size of one, which corresponds to placing each task immediately. The genetic algorithm was not included in this study.

Table A1 (Annexure A) reports the SLA violation rate. Without refinement, batching alone degrades deadline compliance sharply as batches grow, from 0.02 percent with immediate placement to 1.77, 17.2, and 73.2 percent at batch sizes of 10, 20, and 50. Two mechanisms contribute. First, tasks are assigned only when their batch is complete, so an early arrival waits for the rest of the batch while its deadline, measured from arrival, keeps running; for task durations spread evenly between 3 and 14 steps, this waiting alone would cause 0, 5.4, and 37.2 percent violations at batch sizes of 10, 20, and 50, and the remainder reflects queuing. Second, the agent decides for every task in a batch from the same, unchanged view of machine utilization, which tends to send many of them to the same machine: the longest waiting queue grew from 4 tasks with immediate placement to 62 tasks at a batch size of 50.

In Table A1, “Seeds better / worse” counts the seeds in which refinement gave a lower / higher violation rate than the agent alone, and “Longest queue” is the mean over seeds of the longest waiting queue observed with the agent alone.

With NSGA-III refinement, the violation rate was lower than with the agent alone in all ten seeds at batch sizes of 10, 20, and 50 (paired Wilcoxon signed-rank test, exact $p = 0.002$

in each case). Measured against the loss caused by batching, refinement recovered 92, 59, and 26 percent of it at these three sizes, and at a batch size of 10, the refined pipeline reached 0.17 percent, close to the 0.02 percent of immediate placement. Mean energy per task stayed within about 2 percent of the agent alone at these sizes.

The opposite was observed with a batch size of 5. The agent alone remained at 0.15 percent, but refinement raised the mean violation rate to 32.1 percent: seven of the ten seeds exceeded 30 percent, and the other three stayed below 1 percent. Refinement was worse than the agent alone in nine seeds and tied in one (exact $p = 0.004$). Because the unrefined configuration is stable at this size, the collapse is attributable to the refinement stage. We did not identify its cause. One possibility, which we did not test, is that the evaluator scores candidate assignments on an otherwise idle cluster (Section 3.3.1), so that when the agent's own placement is already adequate, the search can move tasks onto machines that are in fact busy.

These results show that the batch size of ten used throughout this paper lies within a narrow working range. The refinement gain reported in the earlier sections is a gain over a batching scheme whose own cost grows quickly with batch size. With small batches, refinement can become a liability. Because this analysis used 10,000 tasks and NSGA-III only, it should be read as a sensitivity analysis of the proposed pipeline and not as a re-estimate of the main comparison.

4.7 Sensitivity to the Number of Reference Points

The reference points used in Sections 4.1 to 4.6 were generated with six divisions per objective, totaling 84 points for four objectives; like the batch size, this value was fixed in advance. To examine its influence, NSGA-III was rerun with three, four, and eight divisions, corresponding to 20, 35, and 165 reference points, using the same ten seeds, a batch size of ten, and the reduced protocol of 10,000 tasks per run. The 84-point setting is the batch-size-ten run of Section 4.6. In the three new settings, reference points decided which candidates survived in 98.9 percent of generations.

Table A2 (Annexure A) reports the results. The mean SLA violation rate ranged from 0.135 to 0.197 percent across the four settings, a spread smaller than the seed-to-seed standard deviation within any single setting (0.11 to 0.19 percentage points). None of the three new settings differed from the 84-point setting in a paired comparison (exact Wilcoxon signed-rank $p = 0.63$, 0.16, and 0.24 for 20, 35, and 165 points), and a Friedman test across the three new settings was borderline ($p = 0.052$). With ten seeds, differences of this size cannot be resolved.

In Table A2, the 84-point row is the batch size of ten from Table A1. The last two columns give the exact paired Wilcoxon signed-rank p -value of each setting against it, and for energy also the mean paired change.

Energy consumption showed a small but systematic dependence. Mean energy per task rose steadily with the number of reference points, from 609.8 at 20 points to 611.0 at 35, 612.2 at 84, and 612.9 at 165. Among the three new settings, energy was higher with more reference points in all ten seeds for each pair of settings (exact $p = 0.002$ for each pair, the smallest value attainable with ten pairs). Relative to the 84-point setting, energy was lower in all ten seeds at 20 and 35 points (exact $p = 0.002$ for each) and higher in nine of ten seeds at 165 points (exact $p = 0.014$). The total difference between 20 and 165 points is 0.5 percent, small relative to the 2.4 percent variation in energy between seeds, and its cause was not investigated.

The tested range does not show a reference-point setting that clearly improves deadline compliance, and the 84-point value used throughout this paper lies within the range where the SLA results are statistically indistinguishable.

4.8 Sensitivity to the Final-Selection Weights

The weights of Equation (4), 0.40, 0.30, 0.20, and 0.10 for SLA, energy, makespan, and utilization, were fixed in advance. In NSGA-III they act only once per batch to choose the single assignment that is deployed from the final population of twenty candidates. To examine their influence, the full pipeline was rerun with three alternative weight sets, using the same ten seeds, a batch size of ten, 84 reference points, and the reduced protocol of 10,000 tasks per run. The reward weights of the placement agent, given in Equation (3), were not varied, because doing so would require retraining the agent.

Before the experiments, the candidates in the final populations were examined directly. Two of the four objectives barely distinguish them: the SLA term is constant (Section 3.3.1), and the range of the makespan term across the candidates of a batch is about 0.0001 in normalized units. The other two objectives, energy and utilization, vary by about 0.012 and 0.056, and they conflict, because an assignment that raises utilization also raises energy. The weights therefore act mainly through the ratio of the energy weight to the utilization weight. With the original weights this ratio is 3, and the original weights selected the candidate with the highest utilization in 99.97 percent of the batches. A weight set with a ratio of 1 selected exactly the same assignment in every batch of every seed, so its results are identical to those of the original weights (Table A3, Annexure A). Sets with ratios of 5 and 7 selected the candidate with the lowest energy in 84 and 99.9 percent of the batches and changed the original choice in 84 and 99.8 percent of them.

Moving the selection toward energy lowered both energy and SLA violations. With a ratio of 7, mean energy per task fell from 612.2 to 580.7 (−5.1 percent; lower in all ten seeds, exact $p = 0.002$), and the mean SLA violation rate fell from 0.167 to 0.061 percent (lower in eight seeds, higher in one, tied in one; exact $p = 0.012$). A ratio of 5 gave similar results: energy fell by 4.5 percent (all ten seeds, $p = 0.002$), and the violation rate was 0.072 percent ($p = 0.020$). The cost was a lower mean CPU utilization, 28.5 and 29.0 percent against 32.7 percent.

The final-selection rule is therefore a far more influential setting than the number of reference points examined in Section 4.7: its effect on energy is about ten times larger. The original weights were meant to give SLA compliance the largest weight, but because the SLA term is constant in the evaluator, the realized selection favors utilization, and the energy-weighted alternatives produced both lower energy use and fewer violations. We did not examine why the higher-utilization assignments are associated with more violations. Because the weights were varied only for NSGA-III, and the comparison with the genetic algorithm in Sections 4.1 to 4.3 used the original weights for both variants, those results should be read with this in mind; in particular, the energy cost attributed to NSGA-III in Section 4.13 may depend in part on this selection rule.

As in Sections 4.6 and 4.7, the study used 10,000 tasks and NSGA-III only. Three weight sets were examined, which is not an exhaustive search, and a weight set chosen on the basis of these results would need to be validated on seeds that were not used here.

4.9 Scalability and Computational Cost

Scalability with the workload was examined by comparing NSGA-III at 10,000 tasks (the reduced protocol of Sections 4.6 to 4.8), 50,000, and 100,000 tasks on the same ten seeds (Table A4, Annexure A). The SLA violation rate and the energy per task barely change with workload size, and no paired comparison between sizes was significant (exact Wilcoxon p between 0.43 and 0.70 for SLA and between 0.19 and 0.56 for energy per task). Because each batch of ten tasks is refined independently, the cost of the refinement stage grows linearly with the number of tasks. The number of machines was not varied: the placement agent was trained for exactly five virtual machines (a 13-dimensional state), so a larger cluster would require retraining it, which we leave to future work.

Computational cost was measured as wall-clock time (see also Section 4.13). Refining one batch takes about 0.11 seconds for GA+Elitism (9.05 minutes per 50,000-task seed) and 0.58 seconds for NSGA-III (48.56 minutes), a 5.4-fold difference. Absolute times depend on the session: the same 10,000-task configuration took between 7 and 11 minutes per seed in different Colab sessions. A profile of the original code on synthetic batches of ten tasks (25 batches, our own machine, 5.9-fold difference) attributed 67.5 percent of NSGA-III's time to non-dominated sorting, 27.8 percent to objective evaluation, and 2.4 percent to reference-point association. NSGA-III evaluates 820 candidates per batch against 401 for the genetic algorithm, so evaluation explains about a factor of two; the rest comes from the sorting, which uses plain Python loops and was not optimized. A vectorized implementation would be expected to reduce this overhead, but we did not test it.

4.10 Validation on Trace-Derived Task Attributes

To check that the findings do not rest on uniformly distributed task sizes, the placement agent alone, GA+Elitism, and NSGA-III were run on tasks derived from the Bitbrains monitoring traces, with the reduced 10,000-task protocol, ten seeds, and the settings used throughout. The traces contain no task records, so a task was defined as a run of consecutive five-minute readings in which a machine's CPU usage exceeded that machine's own mean; its length gave the duration, and its mean CPU and memory use, as fractions of the machine's capacity, gave the demands. This yielded 1,001,288 tasks from 1,239 machines. During the experiment, tasks were drawn at random from this pool and rescaled so that the mean duration (8.5 steps) and the mean resource-time demand per task matched those of the uniform workload, with demands capped at 50 (the smallest possible machine capacity) and durations at 60 steps (7.6 and 2.2 percent of the tasks reached the caps). The shape of the workload therefore differs from the uniform one, with mostly small and short tasks and a few large and long ones, while the average load does not: mean CPU utilization stayed between 30 and 33 percent in all six configurations. The genetic algorithm used the same forecast array as NSGA-III.

Trace-derived tasks were harder for every method (Table A5, Annexure A): the SLA violation rate of the agent alone rose from 1.77 to 4.53 percent, that of GA+Elitism from 0.48 to 2.55 percent, and that of NSGA-III from 0.17 to 1.93 percent (exact paired Wilcoxon $p = 0.002$ for each method). The ranking of the methods was unchanged, and both refinement procedures were better than the agent alone in all ten seeds. However, the advantage of NSGA-III over the genetic algorithm became smaller: with trace-derived tasks its violation rate was 24 percent lower (1.93 versus 2.55 percent; lower in eight seeds and higher in two; exact $p = 0.020$), against 65 percent lower with uniform tasks (0.17 versus 0.48 percent; ten of ten seeds, $p = 0.002$). The

uniform result reproduces the main comparison of Section 4.3 under the reduced protocol.

The energy relation reversed. With uniform tasks NSGA-III used 1.8 percent more energy than the genetic algorithm (exact $p = 0.002$), whereas with trace-derived tasks it used 1.0 percent less ($p = 0.014$) and 1.9 percent less than the agent alone ($p = 0.037$). The energy cost attributed to NSGA-III in Section 4.13 therefore depends on the task distribution, as it does on the final-selection weights (Section 4.8). These tasks are derived from machine utilization rather than recorded as tasks. They are drawn independently, so the temporal clustering of real arrivals is lost, and the placement agent, trained on uniform tasks, was not retrained. The result therefore tests how well the pipeline transfers, not a pipeline trained on this workload.

4.11 Control: Genetic Algorithm with the Search Budget of NSGA-III

Section 3.3.2 noted that NSGA-III evaluates 820 candidates per batch and uses mutation, against 401 and none for the genetic algorithm. To separate the effect of the survivor-selection rule from that of the search effort, a control was run on the 10,000-task protocol with the same ten seeds. It is the NSGA-III procedure of Algorithm A1 with one change: after each generation the 20 candidates with the best weighted score of Equation (4) survive, instead of the non-dominated sorting and reference-point rule. Offspring generation, mutation, the 820 evaluations per batch, and the final selection are identical. The matched genetic algorithm reached a mean violation rate of 2.00 ± 1.94 percent, against 0.167 ± 0.146 for NSGA-III, 0.482 ± 0.281 for the original genetic algorithm and 1.767 ± 0.852 for the agent alone (Table A5). Its energy per task (627.9) and mean CPU utilization (34.8 percent) were higher than in the other three configurations. Giving the weighted-score search more candidates and mutation therefore did not improve it: its mean violation rate was no lower than that of the agent alone, although the spread across seeds was large. A plausible explanation, which we did not test, follows from Section 4.8: the weighted score rewards utilization, so stronger search on it selects busier assignments that the full simulator penalizes, whereas NSGA-III keeps candidates that differ on energy and makespan. The control gives no support to the view that the advantage of NSGA-III over the original genetic algorithm comes from its larger search budget or its mutation, and it indicates that the survivor-selection rule matters. It was run once, at 10,000 tasks, without paired tests.

4.12 Robustness of the Refinement-Mechanism Effect

The main result is that the two refinement procedures, as implemented, differ measurably in outcomes while the forecasting and placement components remain constant across both configurations. Because the procedures also differ in search budget and the deployed assignment is still chosen by a weighted sum (Sections 3.3.2 and 4.8), the difference cannot be attributed to the objective-merging rule alone, although the budget-matched control of Section 4.11 indicates that the larger search budget does not explain it. This contrasts with the elitism comparison in Section 4.1, where an apparent gain at three seeds disappeared once the runs were made comparable. Here, NSGA-III's gain was maintained at ten seeds, and NSGA-III's own SLA rate was unchanged when the workload was doubled (the genetic algorithm was not run again at that size), which gives stronger reason to interpret the effect as a property of the refinement procedure as implemented rather than a statistical fluke from limited sampling.

4.13 The Cost of Diversity: Runtime and Energy Trade-offs

NSGA-III gave better deadline compliance at a modest but statistically significant energy cost (which depended on the task distribution and on the selection weights, Sections 4.8 and 4.10) and a markedly longer computation time than the genetic algorithm. These findings should be read alongside earlier reports that NSGA-III beats its predecessor NSGA-II on multi-objective problems (Latreche et al. [8]). Wall-clock runtime was logged in the same ten Colab sessions that produced Tables 1 and 2: GA+Elitism averaged 9.05 minutes per seed to complete a full 50,000-task run, whereas NSGA-III averaged 48.56 minutes per seed under an identical workload and identical hardware, a roughly 5.4-fold increase. This gap is due mainly to the non-dominated sorting NSGA-III performs on its pool of forty candidates every generation and, to a lesser extent, to evaluating twice as many candidates (Section 4.9). The population of twenty used here is considerably smaller than the populations NSGA-III was designed and tested with on problems of three to fifteen objectives (Deb and Jain [6]) and smaller than the 84 reference points, so whether the SLA gain would grow, shrink, or hold with larger populations or batch sizes remains open.

A 5.4-fold difference in wall-clock time matters little for an offline comparison but more for a scheduler running continuously in a live cluster. Because refinement here operates on batches of ten completed placement decisions, the reported totals imply a per-batch overhead of roughly 0.11 seconds for GA+Elitism against roughly 0.58 seconds for NSGA-III under the workload and hardware tested. Both are comfortably below the seconds-to-minutes granularity at which most cloud schedulers revisit placement decisions, so NSGA-III's extra runtime would not, by itself, rule out its use in this setting. The picture would change for deployments with tighter refinement cycles, larger batch sizes, or clusters large enough that the per-generation cost of sorting candidates into non-dominated fronts and matching them to reference points grows faster than the equivalent weighted-sum comparison used by GA+Elitism. In such settings, the runtime gap would need to be weighed against the SLA and energy gains reported here, and falling back to the faster genetic algorithm when the refinement window is under time pressure may be a more practical compromise.

4.14 Positioning Against Non-Learning Baselines

The comparison with the rule-based schedulers qualifies the main result. Run on the same workloads and deadline rule, Round Robin and FCFS had no deadline violations and used 1 to 3 percent less energy than the pipeline (Table 5), so the advantage of NSGA-III over the genetic algorithm should not be read as an advantage over simple scheduling rules. Two features of the setting probably favor the rules: they place every task on arrival, whereas the pipeline waits for a batch of ten, and the synthetic cluster is lightly loaded (mean CPU utilization about 33 percent), so a rule that spreads tasks evenly seldom runs out of room. Whether the learned placement and refinement stages pay off under heavier load, more heterogeneous demands, or a placement agent trained with a deadline-aware reward is not tested here; the contribution of this paper is the comparison between the two refinement procedures.

4.15 Limitations

Several aspects of the experimental design were fixed by necessity rather than validated systematically; they are acknowledged here as limitations. First, the reward weights in Equation (3) were held at single, hand-selected values throughout. The fitness weights in Equation (4),

the batch size, and the number of reference points were varied only for NSGA-III and only on a reduced 10,000-task protocol (Sections 4.6 to 4.8), and scalability was tested across workload size but not beyond the five-virtual-machine cluster examined here. Reward-weight sweeps, the same analyses for the genetic algorithm, and larger clusters remain necessary before the advantage can be assumed to generalize. Second, the main comparison used task attributes drawn from uniform synthetic distributions; Section 4.10 repeats a reduced version of it on attributes derived from the Bitbrains traces, where the advantage of NSGA-III became smaller, but those tasks are derived from machine utilization rather than recorded as tasks, so confirming the gains on real task records remains an important next step. Third, the batch-level evaluator has a limitation of its own: because its SLA term is constant across candidates (Section 3.3.1), the refinement stage effectively optimizes three objectives rather than four, so the SLA gains reported here should be read as a downstream effect of the other objectives. Fourth, the two refinement procedures differ in search budget and in the use of mutation (Section 3.3.2), and the budget-matched control of Section 4.11 was run only once, at 10,000 tasks, without per-seed paired tests. Fifth, all runs used one trained placement agent, so the seed-to-seed variation reflects the task stream, not variation between training runs. Sixth, every result uses one deadline rule, arrival plus 4.5 times the duration; a tighter rule would penalize the pipeline's batching delay much more heavily (Section 4.6), which was not tested.

5. Conclusion

This paper compared two versions of the periodic refinement stage in a three-stage cloud scheduling pipeline, an elitist genetic algorithm and NSGA-III, with the forecasting and placement stages held fixed throughout. Across ten independent seeds, NSGA-III cut mean SLA violations from 0.495 percent to 0.178 percent, a gap a Wilcoxon signed-rank test confirmed as statistically significant with a large effect size ($p = 0.0039$, $r = 0.854$), and NSGA-III's SLA rate stayed stable when the tested workload was doubled from 50,000 to 100,000 tasks (the genetic algorithm was not run at 100,000 tasks; $p = 0.426$). The gain came with a small but statistically significant rise in mean energy consumption ($p = 0.0020$, $r = 0.886$) and a roughly 5.4-fold longer runtime (48.56 versus 9.05 minutes per seed). These results suggest that, with the forecasting and placement stages unchanged, NSGA-III as implemented yields a repeatable improvement in deadline compliance over the genetic algorithm as implemented; since the two procedures differ in search budget (820 versus 401 candidate evaluations per batch) and the deployed assignment is still chosen by a weighted sum, the improvement cannot be ascribed to Pareto selection alone. A genetic algorithm given the same budget and operators did, however, have a higher mean violation rate (2.00 percent at 10,000 tasks), so extra search does not appear to explain the gain. Sensitivity analyses on a reduced 10,000-task protocol showed a narrow working range for the batch size, little dependence on the number of reference points, and a strong dependence on the final-selection weights: because the SLA term is constant in the evaluator, energy-weighted alternatives lowered both energy use and SLA violations. With task attributes derived from the Bitbrains traces, the ranking held, but NSGA-III's SLA advantage over the genetic algorithm shrank from 65 to 24 percent, and its energy penalty disappeared. Round Robin and FCFS, run under the same deadline rule, recorded no violations and used less energy than either pipeline, so the findings concern the ranking of the two refinement procedures and not an advantage over simple rules. Future work should repeat the matched-budget control at 50,000 tasks with paired tests, measure deadlines from arrival inside the evaluator, and confirm the gain on real task records.

Ethical Statement

This study did not involve human participants, human data, or animal subjects. All experiments were conducted using simulated cloud environments and publicly available, anonymized cluster-monitoring traces.

Conflict of Interest

The authors declare that they have no known conflicts of interest or competing interests that could have influenced the work reported in this paper.

Funding

This research received no external funding from any funding agency in the public, commercial, or not-for-profit sectors.

Acknowledgement

The authors thank the College of Education for Pure Science, University of Wasit, for providing the computing resources used to carry out this research.

References

- [1] Arunarani, A. R., Dhanabalachandran Manjula, and Vijayan Sugumaran. "Task Scheduling Techniques in Cloud Computing: A Literature Survey." *Future Generation Computer Systems* 91 (2019): 407-415.
- [2] Bodra, Deep, and Sushil Khairnar. "Machine Learning-Based Cloud Resource Allocation Algorithms: A Comprehensive Comparative Review." *Frontiers in computer science* 7 (2025): 1678976.
- [3] Mao, Hongzi, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. "Resource Management with Deep Reinforcement Learning." In *Proceedings of the 15th ACM workshop on hot topics in networks*, 2016, 50-56.
- [4] Osama, Muhammad, Syed Shah Sultan Mohiuddin Qadri, Rauf Ahmed Shams Malick, Muhammad Farrukh Shahid, and Kamran Dawood. "Enhanced Task Scheduling in IAAS Cloud Environments Using Elitism-Based Genetic Algorithms." In *2024 8th International Conference on Electrical, Telecommunication and Computer Engineering (ELTICOM)*, IEEE, 2024, 62-67.
- [5] Osama, Muhammad, Syed Shah Sultan Mohiuddin Qadri, Muhammad Farrukh Shahid, and Ahmet Kabarcık. "Enhancing Cloud Architecture's Efficiency with an Adaptive Elitism-Based Genetic Algorithm for Task Scheduling." In *2nd GCC International Conference on Industrial Engineering and Operations Management*. IEOM Society, 2024.
- [6] Deb, Kalyanmoy, and Himanshu Jain. "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems with Box Constraints." *IEEE transactions on evolutionary computation* (2014). 18, no. 4 577–601.
- [7] Wietheger, Simon, and Benjamin Doerr. "A Mathematical Runtime Analysis of the Non-Dominated Sorting Genetic Algorithm III (NSGA-III)." In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence (IJCAI 2023)*,

5657–5665. 2023.

- [8] Imene, Latreche, Slatnia Sihem, Kazar Okba, and Batouche Mohamed. “A Third Generation Genetic Algorithm NSGAIII for Task Scheduling in Cloud Computing.” *Journal of King Saud University Computer and Information Sciences* 34, no. 9 (2022): 7515-7529.
- [9] Tang, Hongtao, Yu Xiao, Wei Zhang, Deming Lei, Jing Wang, and Tao Xu. “A DQL-NSGA-III Algorithm for Solving the Flexible Job Shop Dynamic Scheduling Problem.” *Expert Systems with Applications* 237 (2024): 121723.
- [10] Derrac, Joaquín, Salvador García, Daniel Molina, and Francisco Herrera. “A Practical Tutorial on the Use of Nonparametric Statistical Tests as a Methodology for Comparing Evolutionary and Swarm Intelligence Algorithms.” *Swarm and evolutionary computation* 1, no. 1 (2011): 3-18.
- [11] Carrasco, Jacinto, Salvador García, M. Mar Rueda, Swagatam Das, and Francisco Herrera. “Recent Trends in the Use of Statistical Tests for Comparing Swarm and Evolutionary Computing Algorithms: Practical Guidelines and a Critical Review.” *Swarm and Evolutionary Computation* 54 (2020): 100665.
- [12] Xing, Yanli. “Work Scheduling in Cloud Network Based on Deep Q-LSTM Models for Efficient Resource Utilization.” *Journal of Grid Computing* 22, no. 1 (2024): 36.
- [13] Naaz, Farheen, Ayes Chinmay, and Anmol Pattanaik. “QoS Aware Deep Reinforcement Learning Technique for Task Scheduling in Cloud Computing.” *Discover Computing* 29, no. 1 (2026): 185.
- [14] Zeng, Lei, Qi Liu, Shigen Shen, and Xiaodong Liu. “Improved Double Deep Q Network-Based Task Scheduling Algorithm in Edge Computing for Makespan Optimization.” *Tsinghua Science and Technology* 29, no. 3 (2024): 806-817.
- [15] Zhang, Ziyang, Guodong Fan, Jing Li, Ming Zhu, and Yujie Tian. “MA-DDQN: A Maintenance-Aware Adaptive Double Deep Q-Network for Multi-objective Scheduling in Cloud Manufacturing.” In *International Conference on Intelligent Computing*, Singapore: Springer Nature Singapore, 2025, 123-133.
- [16] Yu, Xiaomo, Jie Mi, Ling Tang, Long Long, and Xiao Qin. “Dynamic Multi Objective Task Scheduling in Cloud Computing Using Reinforcement Learning for Energy and Cost Optimization.” *Scientific Reports* 15, no. 1 (2025): 45387.
- [17] Guo, Wenxia, Wenhong Tian, Yufei Ye, Lingxiao Xu, and Kui Wu. “Cloud resource scheduling with deep reinforcement learning and imitation learning.” *IEEE Internet of Things Journal* 8, no. 5 (2021): 3576-3586.
- [18] Ghaban, Wad, and Hind Salem Alatawi. “Federated Multi-Cloud Task Scheduling with Load Balancing Using Multi-Objective NSGA-II and Reinforcement Learning.” *Scientific Reports* (2026). 23729.
- [19] Guna, G., S. Sarvan, Prashant Johri, Umidbek Abdalov, Sabokhon Erjanova, and Vaibhav C. Gandhi. “Multi-Objective Genetic Algorithms for Dynamic Resource Optimization in Cloud Computing.” In *2025 International Conference on Networks and Cryptology (NETCRYPT)*, IEEE, 2025, 876-881.

- [20] Chen, Jing, Tiantian Du, and Gongyi Xiao. "A Multi-Objective Optimization for Resource Allocation of Emergent Demands in Cloud Computing." *Journal of Cloud Computing* 10, no. 1 (2021): 20.
- [21] Felsy, C., and R. Isaac Sajan. "Actor-Critic Optimization Based Adaptive Task Scheduling Using Deep Double Dueling Q Network for Heterogeneous Cloud Environments." *Transactions on Emerging Telecommunications Technologies* 36, no. 7 (2025): e70185.
- [22] Arbat, Shivani, Vinodh Kumaran Jayakumar, Jaewoo Lee, Wei Wang, and In Kee Kim. "Wasserstein Adversarial Transformer for Cloud Workload Prediction." In *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 11, 2022, 12433-12439.
- [23] Chen, Yang, Jia Hao, Yu Peng, and Hongyan Xia. "Transformer-based Performance Prediction and Proactive Resource Allocation for Cloud-Native Microservices." *Cluster Computing* 28, no. 9 (2025): 568.
- [24] Zhang, Biying, Zhimin Qiu, Yanping Chen, and Yuling Huang. "Cloud Workload Prediction by the De-Based Nonstationary Transformer Model." In *Fourth International Conference on Network Communication and Information Security (ICNCIS 2024)*, vol. 13516, SPIE, 2025, 173-178.
- [25] Zhang, Biying, Guanghao Yang, Feng Yu, and Qinghe Pan. "SWT-CLSTM: A Hybrid Model for Cloud Workload Prediction Combining Smooth Wavelet Transform and Contrastive Learning." *Journal of King Saud University Computer and Information Sciences* 37, no. 10 (2025): 320.
- [26] Ding, Zeyuan, Binbin Feng, and Wangyang Yu. "FELT: Large-Scale Cloud Workload Prediction Through Adaptive Feature-Enhanced and Similarity-Aware Transformer." *Tsinghua Science and Technology* (2025).
- [27] Verma, Abhishek, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale Cluster Management at Google with Borg." In *Proceedings of the tenth european conference on computer systems*, 2015, 1-17.
- [28] Shen, Siqi, Vincent Van Beek, and Alexandru Iosup. "Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters." In *2015 15th IEEE/ACM international symposium on cluster, cloud and grid computing*, IEEE, 2015, 465-474.

Appendix:

Annexure A. Supplementary Material

This annexure collects material that supports the main text but is not needed to follow its argument: the refinement procedure in pseudocode, the overall decision flow, the energy-per-task plot, the headline bar chart, and the sensitivity, scalability, and trace-based tables.

A.1 Refinement Procedure

Algorithm A1: Batch Refinement Procedure (GA+Elitism and NSGA-III)

Input: batch B of 10 DQN-assigned tasks; machine capacities and power profiles C; mode $\in$ {GA, NSGA-III}

Output: refined machine assignment for each task in B

- 1: Initialize population P of 20 candidate assignment vectors, seeded with the original DQN assignment
 - 2: for generation $g = 1$ to 20 do
 - 3: if mode = GA then
 - 4: Score the 20 candidates in P on an idle copy of the cluster with the weighted fitness f of Equation (4), and rank them
 - 5: Keep the best 10 as parents; produce 9 children by single-point crossover of two randomly chosen parents (no mutation)
 - 6: $P \leftarrow$ the 10 parents, the 9 children, and an unchanged copy of the best candidate (elitism)
 - 7: else if mode = NSGA-III then
 - 8: Produce 20 offspring from P by single-point crossover and mutation; evaluate all 40 candidates on an idle copy of the cluster
 - 9: Sort the 40 candidates into Pareto fronts $F_1, F_2, \dots$ by non-domination
 - 10: Associate each candidate with its nearest of 84 reference points
 - 11: Fill the next population front-by-front; break ties using least-crowded reference points
 - 12: $P \leftarrow$ the selected 20 candidates
 - 13: end if
 - 14: end for
 - 15: return the candidate with the best weighted score of Equation (4): the best found over all generations (GA), or the best in the final P (NSGA-III)
-

A.2 Overall Scheduling Flow

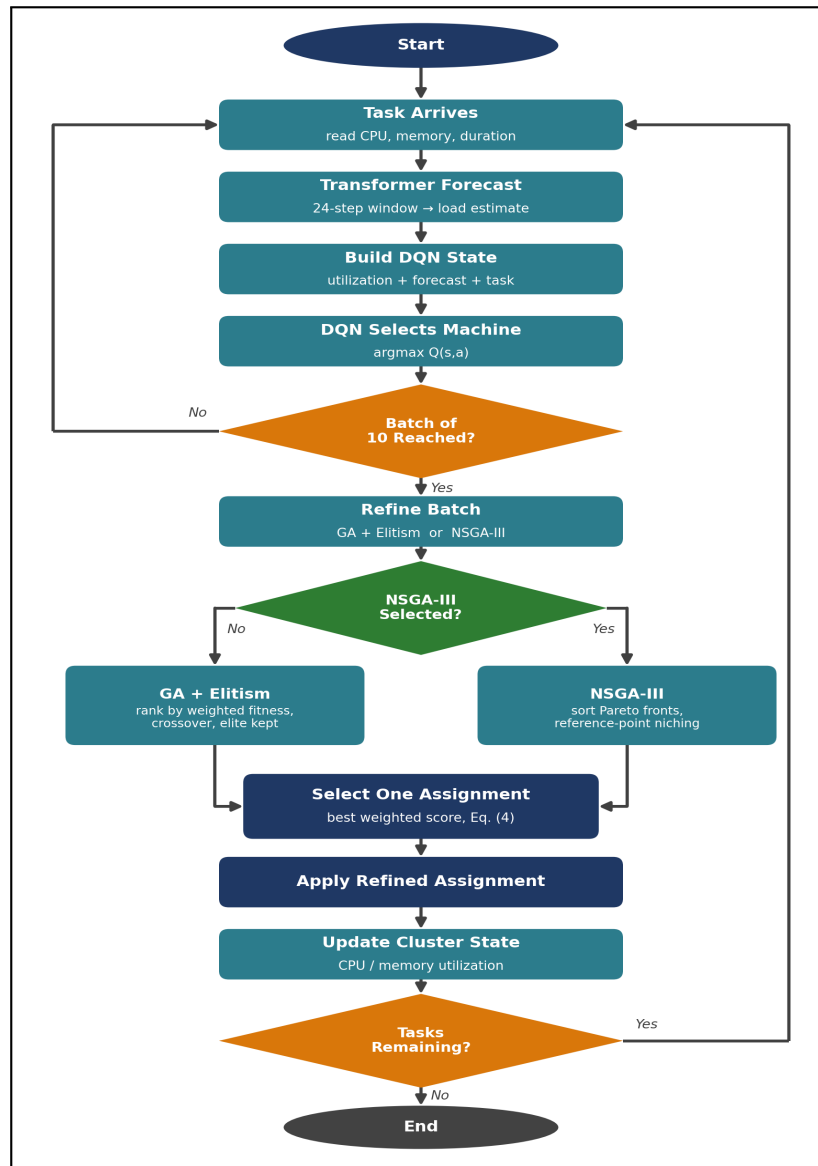


Figure A1. Task scheduling flowchart illustrating where the pipeline branches between GA and NSGA-III refinement. Both branches converge on the selection of one assignment by the weighted score of Equation (4)

A.3 Energy per Task Across Workload Sizes

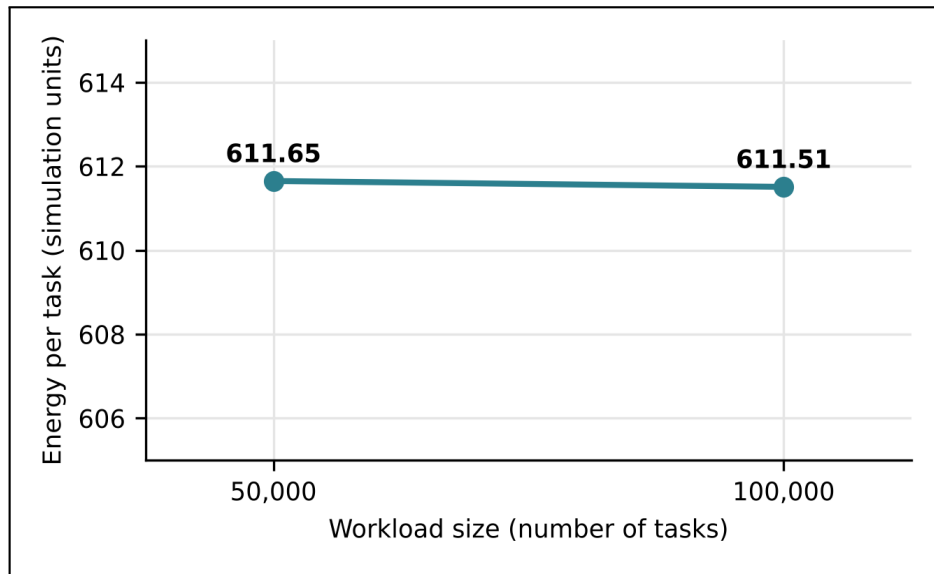


Figure A2. Mean energy per task of NSGA-III at 50,000 and 100,000 tasks (ten seeds). The vertical axis does not start at zero

A.4 Headline Comparison of the Two Refinement Variants

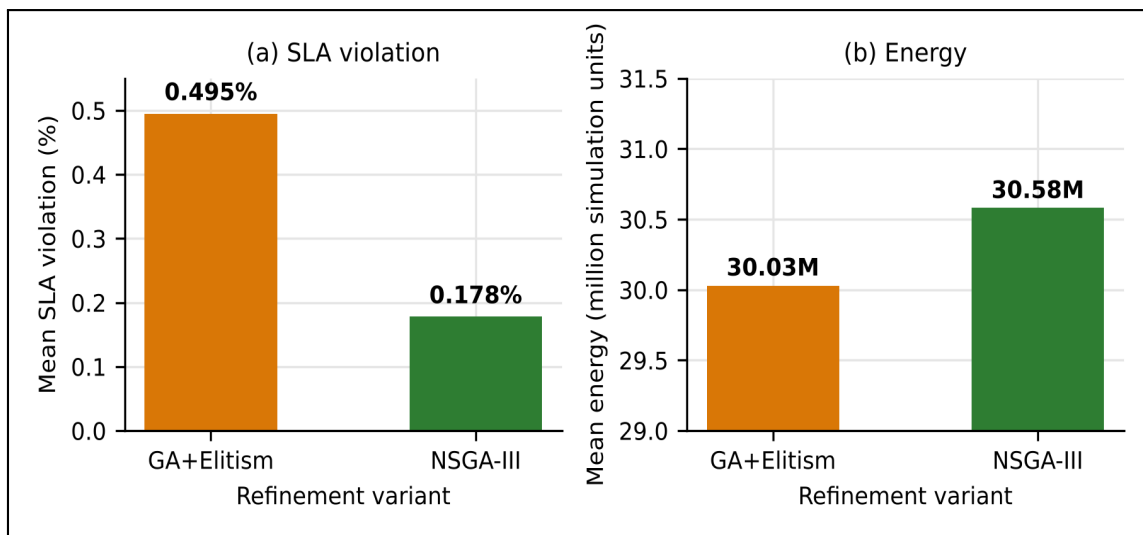


Figure A3. Mean SLA violation rate (a) and mean energy consumption (b) by refinement variant, over ten seeds of 50,000 tasks. The energy axis in panel (b) does not start at zero

A.5 Sensitivity Analyses

Table A1. SLA violation rate (%) of the placement agent alone and with NSGA-III refinement at different batch sizes (10,000 tasks, mean $\pm$ standard deviation (SD) of ten seeds)

Batch size	Agent alone	With NSGA-III	Seeds better / worse	Longest queue
1 (immediate)	0.02 $\pm$ 0.03	—	—	4
5	0.15 $\pm$ 0.12	32.10 $\pm$ 22.69	0 / 9	6
10	1.77 $\pm$ 0.85	0.17 $\pm$ 0.15	10 / 0	12
20	17.21 $\pm$ 3.07	7.06 $\pm$ 0.68	10 / 0	24
50	73.19 $\pm$ 2.80	53.94 $\pm$ 2.97	10 / 0	62

Table A2. NSGA-III performance with different numbers of reference points (10,000 tasks, batch size 10, mean $\pm$ SD of ten seeds)

Divisions	Reference points	SLA violation (%)	Energy per task	SLA vs. 84 points (p)	Energy vs. 84 points (change, p)
3	20	0.161 ± 0.135	609.83	0.63	-0.38%, 0.002
4	35	0.135 ± 0.107	611.02	0.16	-0.19%, 0.002
6	84	0.167 ± 0.146	612.18	—	—
8	165	0.197 ± 0.190	612.87	0.24	+0.11%, 0.014

Table A3. Results for different final-selection weight sets (10,000 tasks, batch size 10, mean $\pm$ SD of ten seeds); E/U is the energy weight divided by the utilization weight

Weight set	Weights (SLA / energy / makespan / utilization)	E/U	SLA violation (%)	Energy per task	CPU utilization (%)
Original (Eq. 4)	0.40 / 0.30 / 0.20 / 0.10	3	0.167 ± 0.146	612.18	32.7
Equal	0.25 / 0.25 / 0.25 / 0.25	1	0.167 ± 0.146	612.18	32.7
Energy-leaning	0.20 / 0.50 / 0.20 / 0.10	5	0.072 ± 0.034	584.61	29.0
Energy-dominant	0.10 / 0.70 / 0.10 / 0.10	7	0.061 ± 0.029	580.72	28.5

A.6 Scalability and Trace-Based Validation

Table A4. NSGA-III performance at three workload sizes (mean $\pm$ SD of ten seeds)

Tasks	SLA violation (%)	Energy per task
10,000	0.167 ± 0.146	612.2
50,000	0.178 ± 0.161	611.6
100,000	0.181 ± 0.161	611.5

Table A5. Results with uniform and trace-derived tasks (10,000 tasks, batch size 10, mean $\pm$ SD of ten seeds)

Tasks	Method	SLA violation (%)	Energy per task	CPU utilization (%)
Uniform	Agent alone	1.767 ± 0.852	599.8	31.0
Uniform	GA+Elitism	0.482 ± 0.281	601.4	31.3
Uniform	NSGA-III	0.167 ± 0.146	612.2	32.7
Trace-derived	Agent alone	4.527 ± 2.770	608.1	31.8
Trace-derived	GA+Elitism	2.551 ± 1.211	602.8	31.2
Trace-derived	NSGA-III	1.927 ± 0.577	596.4	30.4