Journal of Computer Science and Technology Studies
ISSN: 2617-0299 (Online); ISSN: 2708-0099 (Print)
DOI: 10.32996/ijllt
Journal Homepage: www.al-kindipublisher.com/index.php/ijllt

Reinforcement learning-based optimization of distributed data pipelines for cost-efficient and adaptive workload scheduling in cloud data platforms

Gangadhar Reddy Bhimireddy ✉[1], **Bhagam Ravi Teja**[2], **Ramu Mannava**[3], **Ramu Mannava**[4]

[1] *Capgemini, USA*
[2] *Volantsoft Inc, USA*
[34] *Devfi INC, USA*

Corresponding Author: Gangadhar Reddy Bhimireddy**, E-mail:** gangadharsnfdev@gmail.com

ABSTRACT

Distributed data pipelines in cloud data platforms must satisfy deadline commitments while keeping infrastructure spending low, yet most production schedulers still rely on fixed provisioning or hand-tuned threshold rules that react to load only after queues have already grown. This paper presents RL-Sched, a reinforcement learning controller that jointly decides how many compute nodes to provision, whether to acquire them on the on-demand or the preemptible spot market, and which dispatch rule to apply to the waiting pipeline jobs. The scheduling problem is formulated as a Markov decision process whose reward combines infrastructure cost, penalties for missed service level agreements (SLAs), and a small queueing term, and it is solved with a double deep Q-network trained under domain randomization over load, burst, and spot-market conditions. Because evaluating a controller of this kind directly on a production platform is risky, the study used a discrete-time simulator of a cloud data platform with directed-acyclic-graph jobs, diurnal and bursty arrivals, a mean-reverting spot market with price-dependent preemption, and provisioning delays. RL-Sched was compared against two static provisioning policies and two tuned reactive autoscalers over 40 held-out episodes in each of five scenarios. In the nominal scenario, RL-Sched reduced daily cost by 31.2 percent relative to the strongest tuned heuristic (from \$95.0 to \$65.4) and raised average utilization from 49.8 to 68.4 percent, while keeping SLA violations at 0.33 percent. Cost reductions of 33.6 to 52.6 percent were retained under higher load, bursty arrivals, volatile spot prices, and a surge outside the training range, with violation rates between 0.37 and 0.86 percent. Ablation experiments showed that both the spot-market action and the learned dispatch rule contribute materially. The results also exposed a limitation, namely roughly 2.7 times more scaling actions than the heuristic, which motivates switching-cost regularization in future work.

KEYWORDS

Reinforcement Learning, Data Pipelines, Workload Scheduling, Cloud Data Platforms, Autoscaling, Spot Instances, Cost Optimization, Deep Q-Network, Service Level Agreements, Resource Management

ARTICLE INFORMATION

ACCEPTED: 02 June 2026 **PUBLISHED:** 26 July 2025 **DOI:** 10.32996/jcsts.2024.6.3.17

1. Introduction

1.1. Background and Motivation

Modern organizations run much of their analytical and operational reporting on data pipelines, which are chains of extraction, transformation, aggregation, and loading stages executed on shared clusters. A single platform commonly serves hundreds of such pipelines each day, ranging from nightly batch rebuilds of warehouse tables to short micro-batch jobs that refresh feature stores or regulatory extracts. Each pipeline carries an implicit or explicit deadline, because downstream dashboards, models, and reports are only useful if their inputs arrive on time. At the same time, compute is a metered resource. Cloud providers bill by the node-hour, and the difference between a well-utilized and a poorly utilized cluster is directly visible in the monthly invoice [1, 37].

Two characteristics make this setting hard. First, demand is neither constant nor fully predictable. Pipeline arrivals follow daily and weekly cycles, but they are also punctuated by backfills, upstream delays that release many jobs at once, and

Copyright: © 2024 the Author(s). This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC-BY) 4.0 license (https://creativecommons.org/licenses/by/4.0/). Published by Al-Kindi Centre for Research and Development, London, United Kingdom.

month-end processing peaks. Second, the pricing structure offers a choice that interacts with risk. Preemptible capacity, often called spot capacity, is typically sold at a steep discount, but it can be revoked on short notice and its price moves with market conditions [12]. An operator who relies only on on-demand nodes pays a premium for certainty, while an operator who relies heavily on spot nodes accepts rework and deadline exposure.

The prevailing industrial answer is a combination of static capacity planning and reactive autoscaling. Capacity is sized for the expected peak, or a threshold rule adds nodes when utilization or queue length crosses a limit and removes them when the cluster is idle [19, 27]. These policies are simple, interpretable, and easy to audit, which explains their durability. They also have well-known weaknesses. A threshold rule observes only the present, so it cannot anticipate a diurnal ramp or distinguish a transient queue from the start of a sustained surge. A fixed rule cannot weigh the cost of a missed deadline against the cost of an extra node, and it treats the choice of dispatch order as a separate matter from the choice of capacity, even though the two jointly determine whether deadlines are met.

Reinforcement learning (RL) offers a different formulation. Instead of prescribing rules, the operator specifies what is valuable, namely low spending and few missed deadlines, and lets a learning agent discover a policy that trades the two off under the observed dynamics [29]. RL has already produced scheduling policies for packing jobs onto clusters [20] and for scheduling task graphs in data processing systems [21], and it has been explored for virtual machine configuration and cloud autoscaling [8, 26, 30]. Less attention has been given to a combined controller that treats provisioning, market selection, and dispatch as a single decision problem for deadline-bound data pipelines, and that is evaluated for robustness when the workload departs from the conditions seen during training.

1.2. Problem Statement

The problem addressed in this paper can be stated as follows. Given a stream of pipeline jobs, each composed of sequential stages with limited parallelism and a completion deadline, and given a cluster whose size can be changed at the cost of a provisioning delay and whose nodes can be purchased either at a fixed on-demand price or at a volatile and revocable spot price, choose at every control interval a scaling action and a dispatch rule so that the expected total of infrastructure cost and deadline-violation penalties over a day is minimized. The controller must perform well under changes in load intensity, burstiness, and spot-market volatility, including conditions more severe than those encountered during training.

1.3. Research Objectives and Contributions

The study pursued the six objectives listed in Table 1. They span problem formulation, method design, empirical evaluation, and critical analysis of the learned policy.

Table 1: Research objectives

Objective	Description
Obj. 1	Formulate joint node provisioning, purchase-market selection, and job dispatch for deadline-bound data pipelines as a Markov decision process.
Obj. 2	Design a state representation and reward function that make cost and SLA trade-offs explicit and learnable.
Obj. 3	Develop a double deep Q-network controller trained with domain randomization so that it generalizes across workload and market conditions.
Obj. 4	Compare the controller against static provisioning and tuned reactive autoscalers on cost, SLA compliance, latency, utilization, and scaling stability.
Obj. 5	Quantify the contribution of the spot-market action and the dispatch-rule action through ablation experiments.
Obj. 6	Assess robustness under load surges and spot volatility outside the training range and identify the limitations of the learned policy.

The paper makes four contributions. First, it provides a formal model of the combined provisioning, market, and dispatch problem, including an explicit treatment of provisioning delay, preemption rework, and deadline classes. Second, it presents RL-Sched, a double deep Q-network controller with a compact thirteen-dimensional state and a twelve-action decision space designed so that every action maps to an operation that production schedulers already expose. Third, it reports a controlled evaluation against carefully tuned baselines in five scenarios, with paired statistical tests and ablations. Fourth, it documents a failure mode of the learned policy, excessive scaling churn, together with a discussion of how it can be mitigated before deployment.

1.4. Organization of the Paper

Section 2 reviews related work. Section 3 formalizes the problem, and Section 4 describes the proposed controller. Section 5 details the experimental design, and Section 6 reports the results. Section 7 discusses implications and threats to validity, Section 8 outlines limitations and future work, and Section 9 concludes.

2. Related Work

2.1. Scheduling of Data Processing Workloads

Large-scale data processing was popularized by the MapReduce programming model [4] and later by in-memory systems built around resilient distributed datasets [36]. These systems run on clusters managed by general-purpose resource managers such as Borg [34], YARN [33], and Mesos [14]. Early work on cluster scheduling concentrated on fairness and locality. Delay scheduling traded a small wait for better data locality [35], Quincy formulated fair scheduling as a min-cost flow problem [15], and dominant resource fairness extended max-min fairness to multiple resource types [10]. Packing heuristics followed, with Tetris aligning task demands to machine availability across several resource dimensions [11].

Service-level objectives have also been studied directly. Morpheus inferred deadlines for recurring enterprise jobs from historical runs and reserved resources to meet them [16], while Quasar and Paragon used classification to predict how a workload would interfere with others and to choose allocations that satisfy quality-of-service targets [5, 6]. These contributions established that deadline awareness and resource efficiency can be pursued together, but their decision logic is largely heuristic or model-based, and none of them treats the purchase-market choice as part of the scheduling decision.

2.2. Autoscaling and Cost-Aware Provisioning

Elastic provisioning has a long history in cloud computing. Surveys describe threshold-based rules, queueing-theoretic models, control-theoretic approaches, and time-series prediction as the main families of autoscaling techniques [19]. Provisioning under uncertainty has been formulated as a stochastic program that balances reservation and on-demand costs [2], and industrial systems such as Autopilot combine statistical recommenders with machine learning to size jobs automatically [27]. For preemptible capacity, Proteus showed that tiered reliability, with a small stable core and a large transient fringe, allows elastic machine learning training to exploit cheap resources while limiting the cost of revocations [12]. These works support the premise that market-aware elasticity is valuable, and they motivate including a spot-market action in the learned controller.

2.3. Reinforcement Learning for Systems

Reinforcement learning methods learn a policy from interaction rather than from a model of the environment [29]. Value-based deep RL, introduced at scale by the deep Q-network [25], was improved by decoupling action selection from action evaluation to reduce overestimation [32], and policy-gradient methods such as proximal policy optimization offer an alternative with stable updates [28]. In systems research, DeepRM used a policy network to pack jobs with multi-resource demands [20], Decima used graph neural networks to schedule directed acyclic graphs of stages on Spark clusters and reported lower average job completion times than hand-tuned heuristics [21], and device placement for neural network training was framed as a sequential decision problem [24]. In data management, learned optimizers such as Neo and Bao applied RL to query planning [22, 23]. For cloud elasticity, early studies applied Q-learning to resource allocation and virtual machine configuration [8, 26, 30].

Applying RL to production systems nonetheless raises recognized difficulties, including limited exploration opportunities, delayed rewards, partial observability, safety constraints, and sensitivity to implementation details [7, 9, 13]. These issues shape the design choices in this paper, in particular the use of simulation for training, a restricted discrete action space, and an explicit analysis of behavior under distribution shift.

2.4. Research Gap

Table 2 summarizes how representative lines of work relate to the problem studied here. Existing RL schedulers concentrate on task placement within a fixed cluster, while existing autoscalers decide cluster size without learning the interplay with dispatch order or purchase market. The gap addressed by this paper is a single learned controller that operates on all three levers for deadline-bound pipelines and is stress-tested against conditions outside its training distribution.

Table 2: Positioning of the proposed approach relative to representative work

Approach	Learning	Capacity control	Market choice	Dispatch control	Deadline aware
Threshold autoscalers [19]	No	Yes	No	No	No
Autopilot [27]	Statistical	Yes	No	No	No
Proteus [12]	No	Yes	Yes	No	Partly
DeepRM [20]	RL	No	No	Yes	No

Decima [21]	RL	No	No	Yes	No
Morpheus [16]	No	Reservations	No	Yes	Yes
RL-Sched (this paper)	RL	Yes	Yes	Yes	Yes

3. Problem Formulation

3.1. System Model

The platform is modeled in discrete time with a control interval of five minutes, so that a day contains 288 steps. The cluster consists of homogeneous nodes, each offering one processing slot per step. Two pools exist. The on-demand pool contains nodes that are billed at a fixed hourly price and are never revoked. The spot pool contains nodes billed at a fraction of that price, where the fraction follows a stochastic process and a node can be revoked at any step with a probability that rises with the current price fraction. Newly requested nodes become usable only after a provisioning delay of two steps, which corresponds to ten minutes and reflects the time needed to acquire, boot, and register an instance. The cluster size is bounded below by two nodes and above by thirty-two nodes.

3.2. Workload and SLA Model

Jobs represent pipelines. Each job consists of between two and four stages executed in sequence. A stage has a work amount, measured in node-steps, and a maximum parallelism, which is the largest number of slots it can use in a single step. Stage work is drawn from a log-normal distribution with a minimum of one node-step, and parallelism is drawn from the set of one, two, and four slots. The critical path of a job, defined as the minimum number of steps needed to finish its remaining stages with unlimited capacity, determines its deadline. Thirty percent of jobs belong to an urgent class whose deadline equals the critical path multiplied by a factor between 1.4 and 2.4, and the remainder belong to a standard class with a factor between 2.6 and 5.5. Two further steps are added to every deadline to account for dispatch overhead. A job violates its SLA if it has not finished by its deadline, and each job is counted at most once.

The slack of a waiting job at time t is defined as

slack(j, t) = dj − t − cpj(t) (1)

where dj is the deadline of job j and cpj(t) is its remaining critical path. Slack provides the controller with a direct measure of deadline pressure that is more informative than queue length alone.

Arrivals follow a non-homogeneous Poisson process whose mean rate has a sinusoidal daily component with relative amplitude 0.55 around a base rate of 0.62 jobs per step. In addition, a burst process starts with a small probability at each step and multiplies the arrival rate by 2.6 for between eight and eighteen steps, which imitates backfills and delayed upstream releases. With these settings a day contains about 200 jobs.

3.3. Cost Model

Let nod and nsp denote the numbers of on-demand and spot nodes, p the on-demand price per node-hour, mt the spot price fraction, and Δ the step length in minutes. The infrastructure cost incurred in step t is

ct = (nod + nsp · mt) · p · Δ / 60 (2)

with p equal to $0.48 per node-hour. Nodes are billed whether or not they are busy, so idle capacity is a pure loss. The spot price fraction follows a mean-reverting process with occasional upward jumps,

mt+1 = mt + 0.10 (0.35 − mt) + σ εt + Jt (3)

where εt is standard normal noise, σ is the volatility parameter, and Jt is a jump that occurs with a small probability and adds between 0.1 and 0.3 to the fraction. The fraction is clipped to the interval from 0.2 to 0.9. A spot node is revoked in step t with probability

ρt = 0.004 + 0.09 · max(0, mt − 0.3) (4)

so that revocations become more likely exactly when the market is tight. A revocation removes the node immediately and adds one node-step of rework to a randomly chosen running job.

3.4. Markov Decision Process

At each step the controller observes a state vector, selects a pair consisting of a scaling action and a dispatch rule, and receives a reward. The state vector contains thirteen features, listed in Table 3. All features are scaled to roughly the unit range, so no further normalization is required by the learning algorithm.

Table 3: State features observed by the controller

Feature	Description	Scaling
Queue length	Number of jobs waiting or running	Divided by 30
Running jobs	Jobs that received slots in the previous step	Divided by 30
Utilization	Busy slots divided by provisioned nodes in the previous step	0 to 1
On-demand nodes	Current on-demand pool size	Divided by 32
Spot nodes	Current spot pool size	Divided by 32
Pending nodes	Requested nodes not yet available	Divided by 8
Urgent share	Fraction of queued jobs with slack below three steps	0 to 1
Minimum slack	Smallest slack in the queue, truncated to the range of minus five to twenty	Divided by 20
Backlog	Total remaining work of queued jobs, truncated at 300	Divided by 100
Spot price fraction	Current market price as a share of the on-demand price	0.2 to 0.9
Time of day (two features)	Sine and cosine of the daily phase	Minus one to one
Arrival trend	Exponential moving average of arrivals per step	Divided by 1.5

The action space is the Cartesian product of four scaling choices and three dispatch rules, which gives twelve discrete actions. The scaling choices are to hold the current size, to release one idle node, to request one spot node, and to request one on-demand node. The dispatch rules are first-in-first-out (FIFO), earliest deadline first (EDF), and shortest remaining work first (SJF). Restricting scaling to one node per step keeps the action space small and bounds the damage a single poor decision can do, and the decisions compound over many steps to produce large changes in capacity. Table 4 summarizes the action space.

Table 4: Action space of the controller

Component	Options	Operational meaning
Scaling	Hold	Keep the provisioned capacity unchanged
Scaling	Release	Remove one idle node if at least one is idle, preferring on-demand nodes when more than two are present
Scaling	Add spot	Request one spot node, available after the provisioning delay
Scaling	Add on-demand	Request one on-demand node, available after the provisioning delay
Dispatch	FIFO	Serve jobs in arrival order
Dispatch	EDF	Serve jobs with the earliest deadline first
Dispatch	SJF	Serve jobs with the least remaining work first

The reward at step t penalizes spending, missed deadlines, and long queues,

$$r_t = -\left(c_t + \beta\, v_t + \eta\, q_t \right) \quad (5)$$

where v_t is the number of jobs that newly violate their deadline in step t, q_t is the number of waiting jobs, β equals 2.0 dollars per violation, and η equals 0.002 dollars per waiting job and step. The value of β places a missed deadline at roughly five times the cost of running a ten-node on-demand cluster for one step, which expresses a strong but finite preference for meeting deadlines. The objective is to find a policy that maximizes the expected discounted return over an episode of 288 steps.

4. Proposed Method: RL-Sched

4.1. Framework Architecture

Figure 1 shows the closed loop of the proposed controller. Jobs from the workload layer enter the pipeline queue, and the compute pools execute them. At every control interval a state encoder summarizes the queue and cluster into the feature vector of Table 3, the scheduling agent maps it to an action, and the action is applied to the queue through the dispatch rule and to the pools through the scaling operation. The resulting cost and violations form the reward that trains the agent.

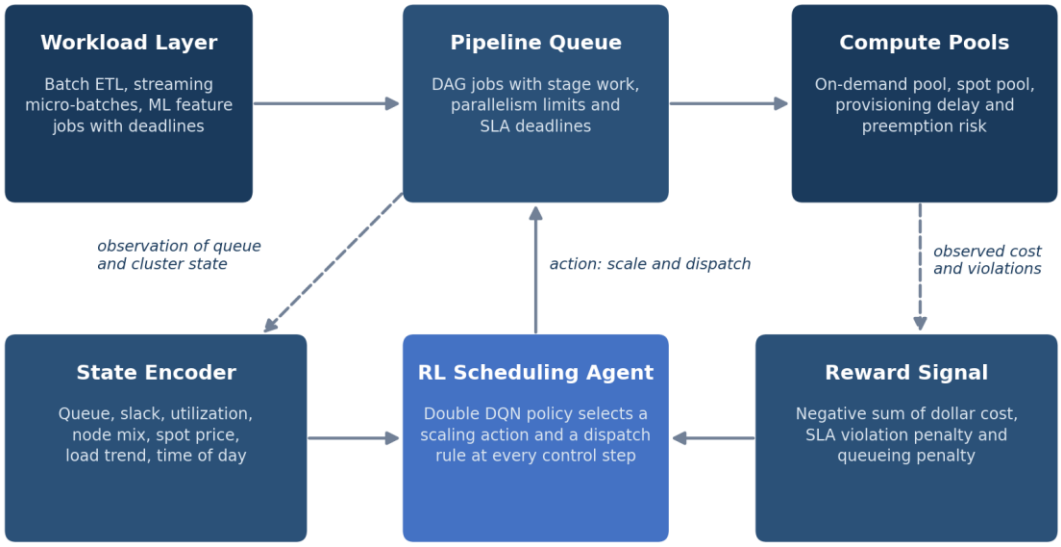


Figure 1: Closed-loop architecture of the proposed RL-based pipeline scheduler

4.2. Double Deep Q-Network

The controller approximates the action-value function with a feed-forward neural network that has two hidden layers of 64 rectified linear units and twelve outputs, one for each action. Learning follows the double deep Q-network scheme [25, 32]. An online network with parameters θ selects the greedy action in the next state, while a slowly updated target network with parameters θ' evaluates it, which reduces the overestimation bias of standard Q-learning. The regression target is

$$y_t = r_t + \gamma\, Q_{\theta'}(s_{t+1}, \operatorname{argmax}_a Q_{\theta}(s_{t+1}, a)) \quad (6)$$

and the network is trained to minimize the Huber loss between $Q_\theta(s_t, a_t)$ and y_t on mini-batches sampled from a replay buffer [18]. The Huber loss limits the influence of rare large penalties, such as a cluster of simultaneous deadline violations, on the gradient. Optimization uses the Adam algorithm [17], and the target network is updated by Polyak averaging with a coefficient of 0.01 after each learning step.

4.3. Training with Domain Randomization

A controller trained on a single set of workload parameters would likely learn that particular environment and fail when load or market behavior changes. To encourage generalization, each training episode draws its parameters from a distribution, a technique known as domain randomization [31]. The load multiplier is drawn uniformly between 0.8 and 1.35, the burst probability between 0.006 and 0.025 per step, the spot volatility between 0.02 and 0.05, and the spot jump probability between 0.005 and 0.02. Exploration follows an epsilon-greedy schedule that decays linearly from 1.0 to 0.05 over the first half of training. Algorithm 1 summarizes the procedure.

Algorithm 1: Training of RL-Sched

```
Initialize online network Q, target network Q', replay buffer B
for episode = 1 to N do
  draw environment parameters (load, bursts, spot volatility) at random
  observe initial state s
  for step t = 1 to 288 do
    with probability epsilon pick a random allowed action, else a = argmax Q(s, .)
    apply a to the cluster; observe reward r and next state s'
    store (s, a, r, s', done) in B
    every 4 steps: sample a mini-batch from B, compute double-DQN targets,
      take one Adam step on the Huber loss, and soft-update Q'
  end for
end for
```

4.4. Deployment Considerations

Three design decisions aim at making the controller adoptable in a production setting. First, the action space contains only operations that existing schedulers and autoscaling interfaces already support, so the learned policy can be introduced behind a feature flag and compared with the incumbent rule. Second, an action mask can remove any action, which allows an operator to forbid spot purchases for compliance reasons or to freeze the dispatch rule; the ablation study in Section 6.5 evaluates exactly these restricted variants. Third, the learned policy is a small network whose inference takes microseconds, so it can run on every control interval without noticeable overhead. A hard guardrail that falls back to the incumbent heuristic when the queue exceeds a safety limit is recommended, although it was not part of the experiments reported here.

5. Experimental Setup

5.1. Simulator

All experiments were conducted in a purpose-built simulator that implements the system, workload, and cost models of Section 3. A simulator was chosen because training and fairly comparing policies on a live platform would expose real pipelines to the exploration of an untrained agent and would not allow identical workloads to be replayed for every method. Table 5 lists the simulator parameters. The workload generator was parameterized to produce utilization levels and burst behavior that are typical of analytical clusters, but it was not calibrated to a specific production trace, a limitation discussed in Section 8.

Table 5: Simulator parameters

Parameter	Value
Control interval and episode length	5 minutes and 288 steps (24 hours)
On-demand price	$0.48 per node-hour
Spot price fraction	Mean 0.35, reversion rate 0.10, bounds 0.2 to 0.9, jumps of 0.1 to 0.3
Spot revocation probability	0.004 + 0.09 × max(0, fraction − 0.3) per node and step
Provisioning delay	2 steps (10 minutes)
Cluster size bounds	2 to 32 nodes; initial size 8 on-demand nodes (the learned agent also starts with 2 spot nodes; static policies start at their fixed size)
Base arrival rate and daily amplitude	0.62 jobs per step and 0.55
Burst process	Start probability 0.012 per step, length 8 to 18 steps, rate multiplier 2.6
Job structure	2 to 4 sequential stages; stage work log-normal (μ = 0.9, σ = 0.55), minimum 1; parallelism from {1, 2, 4, 4}
Deadline factors	Urgent (30 percent of jobs) 1.4 to 2.4; standard 2.6 to 5.5; plus 2 steps
Reward weights	β = 2.0 dollars per violation; η = 0.002 dollars per waiting job and step

5.2. Scenarios

Policies were evaluated in five scenarios, defined in Table 6. The nominal scenario uses the central parameter values. The high-load, bursty, and volatile-spot scenarios each stress one factor within or slightly beyond the training range. The out-of-distribution (OOD) surge scenario combines a load multiplier of 1.7, which exceeds the maximum of 1.35 seen in training, with a high burst probability, and it therefore tests extrapolation. The volatile-spot scenario uses a volatility of 0.06, which also exceeds the training maximum of 0.05.

Table 6: Training and evaluation scenarios

Scenario	Load multiplier	Burst probability	Spot volatility	Spot jump probability
Training (randomized)	0.80 to 1.35	0.006 to 0.025	0.02 to 0.05	0.005 to 0.02
Nominal	1.00	0.012	0.03	0.010
High load	1.35	0.012	0.03	0.010
Bursty	1.00	0.030	0.03	0.010
Volatile spot	1.00	0.012	0.06	0.030
OOD surge	1.70	0.020	0.03	0.010

5.3. Baselines

Four baselines were used. Static-Peak provisions 24 on-demand nodes for the whole day, a size chosen to cover the nominal daily peak, and dispatches in FIFO order. Static-Mean provisions 10 on-demand nodes, which approximates the average demand. Reactive-OD is a threshold autoscaler that adds an on-demand node when the queue exceeds the number of nodes or utilization exceeds 90 percent, removes an idle node when utilization falls below 50 percent and the queue is empty, and dispatches in FIFO order. Reactive-Spot-EDF is a cost-aware variant that scales out when the queue exceeds the number of nodes or utilization exceeds 80 percent, prefers spot nodes when the market fraction is at most 0.5 and no more than 35 percent of waiting jobs are urgent, switches to on-demand nodes otherwise, releases an idle node when utilization falls below 60 percent and the queue is empty, and dispatches by earliest deadline.

To avoid weak baselines, the two reactive policies were tuned. A grid of 72 configurations, formed by the queue threshold, the utilization thresholds, the use of spot nodes, and the dispatch rule, was evaluated on eight episodes drawn from the training distribution, and the configuration with the lowest sum of daily cost and four dollars per percentage point of violations was retained for each family. The tuning episodes used seeds that do not overlap with the evaluation seeds.

5.4. Evaluation Metrics

Seven metrics were recorded for every episode. Daily cost is the sum of per-step infrastructure cost. The SLA violation rate is the percentage of jobs that missed their deadline. Mean and 95th percentile latency are measured from arrival to completion for finished jobs. Utilization is the share of provisioned node-steps that processed work, and spot share is the proportion of node-steps provided by spot nodes. Scaling events counts the number of scale-up and scale-down operations in a day and serves as a measure of operational churn.

5.5. Implementation and Hyperparameters

The agent and the simulator were implemented in Python, with the neural network and the Adam optimizer written directly with NumPy so that every step of learning is transparent. Table 7 lists the hyperparameters. Each variant was trained from three random initializations for 500 episodes, which corresponds to 144,000 interaction steps per initialization. Training one agent took under one minute on a single CPU core, which illustrates that the approach does not require specialized hardware.

Table 7: Hyperparameters of the double DQN agent

Hyperparameter	Value
Network architecture	13 inputs, two hidden layers of 64 ReLU units, 12 outputs
Optimizer and learning rate	Adam, 7 × 10-4
Discount factor	0.99
Mini-batch size and replay capacity	64 and 60,000 transitions
Learning frequency	Every 4 environment steps after 1,000 stored transitions
Target network update	Polyak averaging, coefficient 0.01 per learning step
Loss	Huber loss (gradient clipped at 1)
Exploration	Epsilon decays linearly from 1.0 to 0.05 over 250 episodes
Training length	500 episodes per initialization, three initializations per variant

5.6. Evaluation Protocol

Each method was evaluated on 40 episodes in each scenario, using seeds that do not occur in training or tuning. All methods saw the same 40 workloads and market paths in a given scenario, a design known as common random numbers, which reduces the variance of paired comparisons. For the learned controller, episode i used the agent from initialization i mod 3, so each of the three agents ran about a third of the episodes. Statistical comparisons use paired t-tests and Wilcoxon signed-rank tests on per-episode costs, and effect sizes are reported as Cohen's d for paired samples.

6. Results and Analysis

6.1. Training Behavior

Figure 2 shows the smoothed episode return during training for the full controller and for two restricted variants. The full controller improved from an average return of −132.9 over its first 50 episodes to −73.1 over its last 50 episodes, averaged across initializations. The variant without the dispatch choice reached −80.1, and the variant without the spot action reached −123.8. Returns during training are noisy by construction, because each episode draws different environment parameters and because exploration continues at the lowest rate. The shaded bands, which show the spread across initializations, remained narrow for the full controller, which suggests that learning was not unusually sensitive to the random seed in this setting.



Figure 2: Training curves of the full controller and two restricted variants (mean and standard deviation over three initializations)

6.2. Main Results in the Nominal Scenario

Table 8 reports the nominal-scenario results. Static-Peak met every deadline but cost $276.5 per day at an average utilization of 30.8 percent, which shows how much capacity a peak-sized cluster leaves idle. Static-Mean was cheap at $115.2 but violated 23.4 percent of deadlines, and its 95th percentile latency exceeded two hours. The tuned Reactive-OD policy reduced cost to $217.4 without missing deadlines, but it still held 18.9 nodes on average. The tuned Reactive-Spot-EDF policy was a considerably stronger baseline at $95.0 with no violations, because it used spot capacity for 75.7 percent of node-steps and dispatched by deadline.

Table 8: Results in the nominal scenario (mean ± standard deviation over 40 episodes)

Method	Cost (USD/day)	SLA viol. (%)	Mean lat. (min)	P95 lat. (min)	Util. (%)	Spot share (%)	Scale events
Static-Peak	276.5 ± 0.0	0.00 ± 0.00	29.3 ± 1.3	55.3 ± 4.2	30.8 ± 4.2	0.0 ± 0.0	0 ± 0
Static-Mean	115.2 ± 0.0	23.42 ± 19.08	71.7 ± 34.6	144.0 ± 73.2	73.2 ± 9.1	0.0 ± 0.0	0 ± 0
Reactive-OD	217.4 ± 36.1	0.02 ± 0.11	29.9 ± 1.2	56.1 ± 4.1	39.5 ± 3.9	0.0 ± 0.0	52 ± 12
Reactive-Spot-EDF	95.0 ± 19.4	0.00 ± 0.00	31.9 ± 1.6	65.6 ± 7.1	49.8 ± 3.6	75.7 ± 11.1	75 ± 13
RL-Sched	65.4 ± 13.7	0.33 ± 0.61	37.3 ± 5.5	82.9 ± 20.0	68.4 ± 16.0	80.1 ± 4.1	202 ± 19

RL-Sched achieved a daily cost of $65.4, which is 31.2 percent below Reactive-Spot-EDF, 69.9 percent below Reactive-OD, and 76.3 percent below Static-Peak. It did so by running a leaner cluster, with 11.5 nodes on average compared with 15.7

for Reactive-Spot-EDF, and by using spot nodes for 80.1 percent of node-steps. Utilization rose to 68.4 percent. The price of the saving was a small increase in violations to 0.33 percent and a slower average completion, with a mean latency of 37.3 minutes compared with 31.9 minutes, an increase of about 17 percent. Figure 3 places all methods in the cost and violation plane. These changes are the expected consequence of the reward, which accepts occasional deadline misses when avoiding them would require paying for capacity that is idle most of the time.

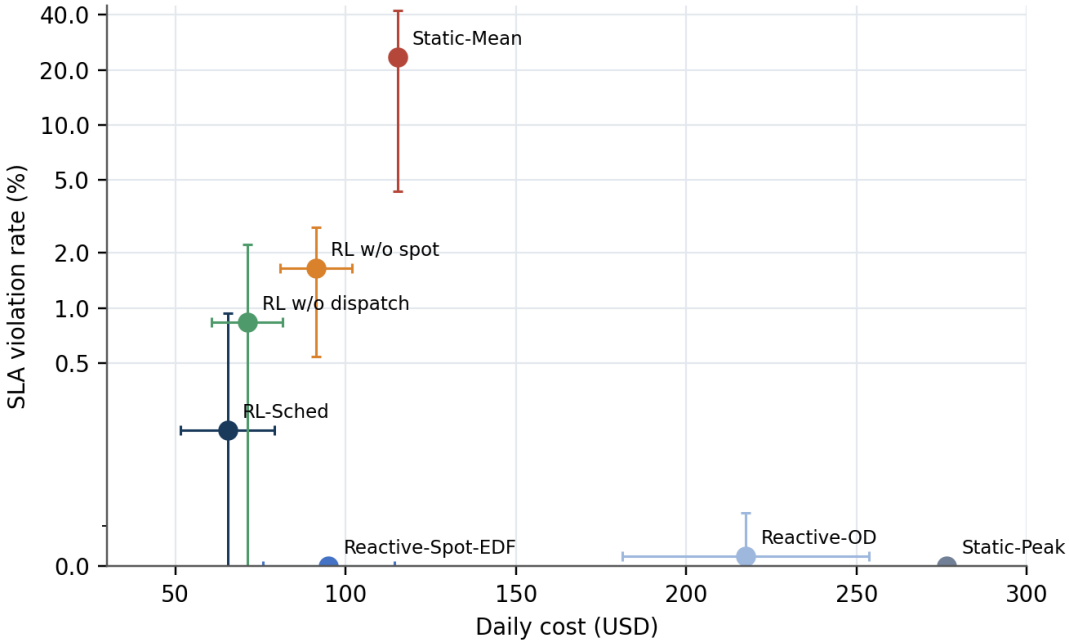


Figure 3: Cost and SLA violation rate of all methods in the nominal scenario (points show means, bars show one standard deviation; the vertical axis is symmetric logarithmic)

6.3. Robustness Across Scenarios

Table 9 and Figure 4 report cost and violations in all five scenarios. The cost advantage of RL-Sched over the strongest heuristic was 39.9 percent under high load, 33.6 percent under bursty arrivals, 52.6 percent when spot prices were volatile, and 41.0 percent in the out-of-distribution surge. The largest gain occurred in the volatile-spot scenario, where the heuristic's cost rose from $95.0 to $147.6 which is consistent with its rule of buying on-demand nodes whenever the price fraction exceeds 0.5, so that its spot share fell to 40.1 percent, whereas the cost of RL-Sched changed little. Violations of RL-Sched stayed between 0.33 and 0.86 percent, and they were highest in the surge scenario, in which load exceeded the range seen in training by more than 25 percent.

Table 9: Cost and SLA violations by scenario (mean over 40 episodes)

Scenario	Static-Peak cost	Reactive-OD cost	Reactive-Spot-EDF cost	RL-Sched cost	Cost reduction vs. best heuristic	Viol. heuristic (%)	Viol. RL-Sched (%)
Nominal	276.5	217.4	95.0	65.4	31.2%	0.00	0.33
High load	276.5	269.6	123.3	74.1	39.9%	0.00	0.52
Bursty	276.5	256.5	106.2	70.5	33.6%	0.00	0.37
Volatile spot	276.5	220.0	147.6	69.9	52.6%	0.00	0.52
OOD surge	276.5	301.8	147.7	87.1	41.0%	0.00	0.86

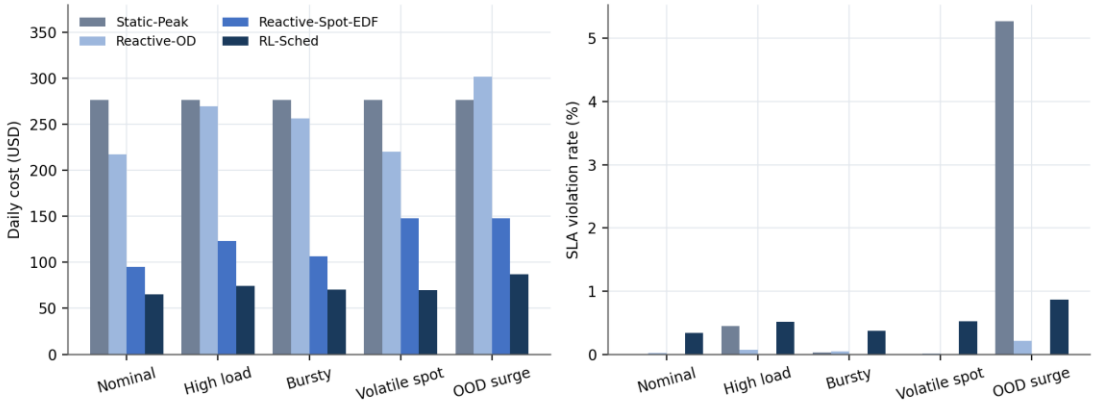


Figure 4: Daily cost (left) and SLA violation rate (right) of four methods across the five evaluation scenarios

Static policies illustrate why provisioning for a fixed size is fragile. Static-Peak, which was adequate in the nominal scenario, violated 5.3 percent of deadlines in the surge scenario, and Static-Mean violated 74.8 percent. Reactive-OD protected deadlines in every scenario, with at most 0.22 percent violations, but its cost reached $301.8 in the surge scenario because its on-demand scale-out was never offset by cheaper capacity.

6.4. Temporal Behavior

Figure 5 follows one bursty day for Reactive-Spot-EDF and RL-Sched. The top panel shows the expected arrival rate, which contains several bursts. During the largest burst, between about hours ten and twelve, the heuristic grew to 32 nodes, the cluster maximum, and stayed at 25 nodes or more for roughly four hours, a pattern that is consistent with its scale-down condition requiring an empty queue. RL-Sched peaked near 22 nodes and released capacity within about two hours. The learned policy tolerated a larger queue than the heuristic during the afternoon bursts, with up to 19 waiting jobs, which is consistent with its use of slack and dispatch order to protect urgent jobs while deferring work that still has time.

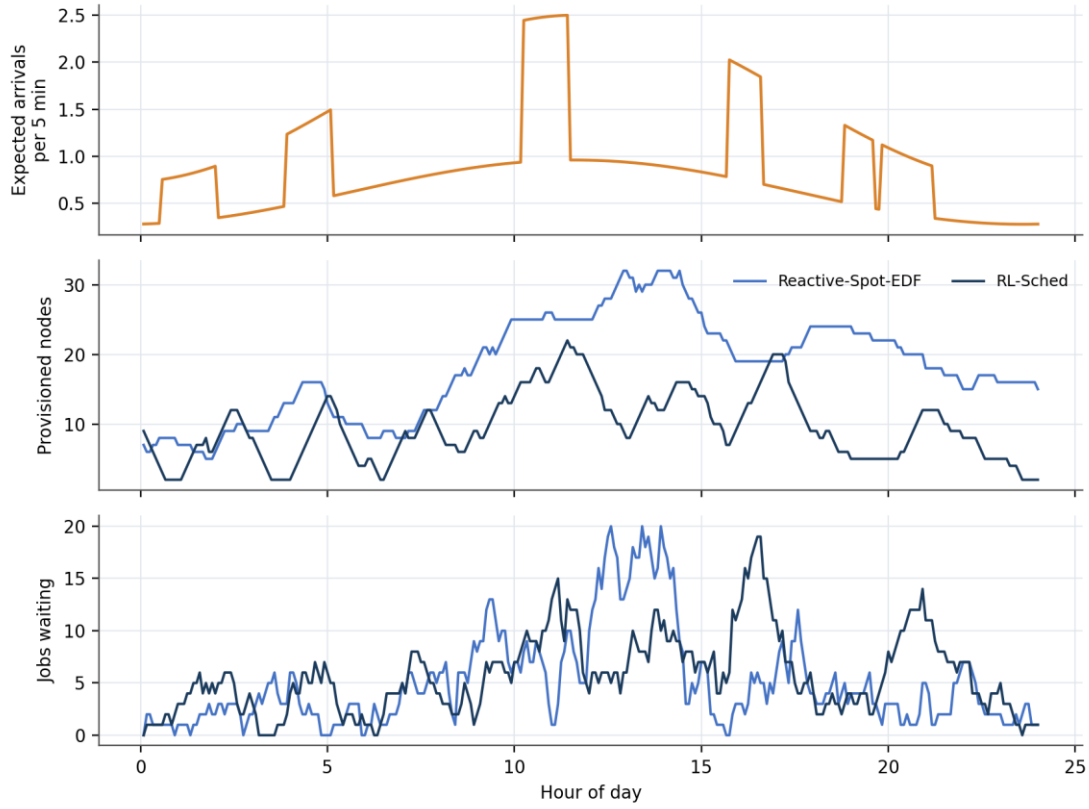


Figure 5: Arrival rate, provisioned nodes, and queue length over one bursty day for Reactive-Spot-EDF and RL-Sched

The trace also reveals the main weakness of the learned policy. The node count of RL-Sched oscillates with a saw-tooth pattern, which reflects frequent alternation between adding a spot node and releasing a node. Table 8 quantifies this behavior, with 201.7 scaling operations per day for RL-Sched compared with 74.7 for Reactive-Spot-EDF. Because the

simulator charges no direct fee for a scaling action, the agent had no reason to avoid it. In a real platform every provisioning request has operational overhead, so the behavior would need to be damped, as discussed in Section 7.

6.5. Ablation Study

Two restricted variants were trained to isolate the contribution of individual action components. The no-spot variant could only add on-demand nodes, and the no-dispatch variant was restricted to FIFO. Table 10 and Figure 6 show that removing the spot action increased cost by 39.6 percent in the nominal scenario, from \$65.4 to \$91.3, and raised violations from 0.33 to 1.65 percent. The no-spot controller compensated by running its cluster at 91.0 percent utilization, which left no headroom and lengthened the mean latency to 53.1 minutes. Removing the dispatch choice had a smaller effect in the nominal scenario, with cost rising to \$71.1 and violations to 0.84 percent, but a much larger one in the surge scenario, where violations rose from 0.86 to 3.64 percent. Dispatch flexibility therefore functions mainly as a protection against overload, because it lets the controller sacrifice the completion time of long jobs to save short and urgent ones.

Table 10: Ablation results (mean over 40 episodes)

Variant	Nominal cost	Nominal viol. (%)	Volatile spot cost	Volatile spot viol. (%)	OOD surge cost	OOD surge viol. (%)
RL-Sched (full)	65.4	0.33	69.9	0.52	87.1	0.86
Without dispatch choice	71.1	0.84	76.3	1.10	116.8	3.64
Without spot action	91.3	1.65	88.9	1.72	165.9	1.80

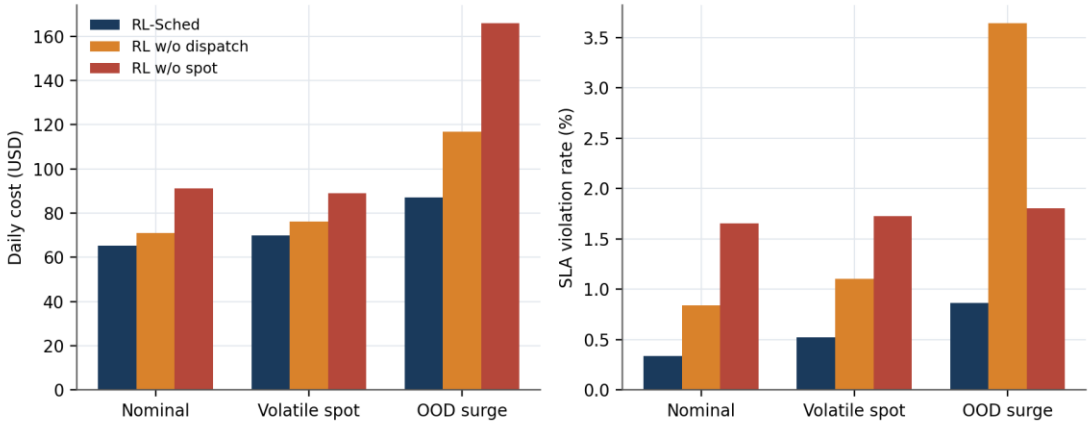


Figure 6: Cost (left) and SLA violation rate (right) of the full controller and the two restricted variants

6.6. Statistical Analysis

Table 11 reports paired comparisons of daily cost between RL-Sched and Reactive-Spot-EDF. In every scenario the mean paired difference was negative and the 95 percent confidence interval excluded zero. Both the paired t-test and the Wilcoxon signed-rank test gave p-values below 0.001, and the effect sizes were large, with Cohen's d between 1.31 and 2.34 in absolute value. The comparison of episode return, which combines cost and penalties, led to the same conclusion, with differences in favor of RL-Sched between 27.7 and 74.5 dollars per day and p-values below 0.001. These tests treat the 40 evaluation episodes as independent samples and the three trained agents as a pooled population, so they describe the average behavior of the training procedure and not that of a single trained agent.

Table 11: Paired comparison of daily cost, RL-Sched minus Reactive-Spot-EDF

Scenario	Mean difference (USD)	95% CI (USD)	t statistic	p-value	Cohen's d
Nominal	−29.7	−36.9 to −22.4	−8.32	< 0.001	−1.31
High load	−49.2	−58.9 to −39.5	−10.23	< 0.001	−1.62
Bursty	−35.7	−44.2 to −27.2	−8.49	< 0.001	−1.34
Volatile spot	−77.7	−88.3 to −67.0	−14.79	< 0.001	−2.34
OOD surge	−60.6	−69.1 to −52.0	−14.33	< 0.001	−2.27

6.7. Interpretation of the Learned Policy

Inspecting the actions taken by a trained agent over ten nominal episodes gives insight into the strategy it adopted. Table 12 shows that the agent released a node in 63.3 percent of steps and requested a spot node in 35.5 percent, whereas it

almost never held the capacity constant or requested an on-demand node, with shares of 0.8 and 0.4 percent. It preferred SJF dispatch in 59.0 percent of steps and EDF in 37.4 percent, using FIFO only 3.6 percent of the time. The policy is thus a tracking strategy that keeps capacity close to current demand by continuously adding cheap nodes and releasing idle ones, and it combines this with a dispatch rule that favors short jobs while falling back to deadline order when slack is tight. The preference for SJF is consistent with scheduling theory, since serving short jobs first minimizes average waiting time, and the reward penalizes waiting through both the queue term and the deadline term.

Table 12: Share of actions chosen by a trained agent in the nominal scenario

Action component	Option	Share of steps (%)
Scaling	Hold	0.8
Scaling	Release one node	63.3
Scaling	Add spot node	35.5
Scaling	Add on-demand node	0.4
Dispatch	FIFO	3.6
Dispatch	EDF	37.4
Dispatch	SJF	59.0

7. Discussion

7.1. Practical Implications

The results indicate that a compact learned controller can lower the cost of running deadline-bound pipelines substantially relative to well-tuned reactive rules, mainly by combining three behaviors. It keeps utilization higher through finer capacity tracking, it uses cheap preemptible capacity more consistently without being undone by preemption risk, and it chooses the dispatch order according to current deadline pressure. The comparison with static policies is a reminder of how much of the cost in many platforms is idle capacity. Even a good reactive autoscaler left half of its provisioned node-steps unused in the nominal scenario.

The cost advantage must be read together with the service trade-off. The heuristic achieved zero violations, whereas RL-Sched allowed 0.33 to 0.86 percent. Whether this exchange is acceptable depends on the penalty attached to a missed deadline, which is encoded in the value of β. An operator for whom a missed regulatory extract has a high business cost would raise β, which pushes the learned policy toward more headroom, and the training procedure can be rerun for any chosen value. The value of using learning here is that the trade-off becomes an explicit, tunable parameter, whereas in threshold rules it is hidden within several interacting thresholds.

7.2. Mitigating Scaling Churn

The high number of scaling operations is the most important practical concern. Several remedies are available without changing the overall approach. A switching cost can be added to the reward so that each add or release action incurs a small penalty. The scaling choice can be made less frequent by letting the agent act every few steps, or by adding a cooldown after any change. The action set can also be extended with a hold-for-longer option or with scaling by several nodes at once. A related and simple safeguard is to apply a minimum dwell time to released nodes, which mirrors the minimum billing increments that many providers use. These modifications would need to be evaluated, and it is possible that part of the cost advantage would be lost in exchange for stability.

7.3. Threats to Validity

Several threats to validity apply. The most important is that the evaluation is based on a simulator with a synthetic workload. The simulator implements the mechanisms that matter for the decision problem, but real clusters have additional phenomena, such as data skew, shuffle costs, heterogeneous instance types, multi-tenancy interference, and correlated failures, which the model omits. The absolute costs reported here should therefore not be transferred to a particular platform, and the relative differences between methods should be treated as evidence of what the controller can learn and not as a forecast of savings. Second, the baselines, although tuned, are a limited set, and they do not include predictive autoscalers that forecast demand from history. Third, only one RL algorithm and one network size were examined, and results for policy-gradient methods or recurrent networks may differ. Fourth, the experiments used three random initializations, which is a small number given the known variance of deep RL training [13]. The narrow spread observed across initializations is encouraging, but it does not substitute for a larger replication.

8. Limitations and Future Directions

The study has limitations beyond those of the simulator. The state of the controller does not include information about the data volume each job processes or about its historical runtime, so it cannot anticipate that a particular pipeline will take longer than its stage estimates suggest. Each stage in the model has known work, which is a simplification, because in

practice runtimes are estimated from earlier executions and the estimates carry error. The action space, while practical, restricts scaling to one node per decision, and it includes only one instance type. The reward uses a fixed penalty for every violation, whereas contracts often distinguish between classes of SLA with different penalties.

Future work will proceed in five directions. The first is calibration and validation against traces from a production platform, including replaying recorded arrivals and market prices, followed by a shadow deployment in which the controller recommends actions without executing them. The second is the regularization of scaling behavior through switching costs and temporal abstraction, as discussed in Section 7.2. The third is the extension of the state with predicted runtimes and forecast arrivals, and of the action space with multiple instance families and reserved capacity. The fourth is the study of safe RL techniques, for example constrained optimization with an explicit violation budget [9], which would allow an operator to specify a target violation rate instead of a penalty weight. The fifth is the evaluation of alternative learning algorithms, including policy-gradient and actor-critic methods, and of larger replication studies with more random initializations.

9. Conclusion

This paper examined whether reinforcement learning can improve the economics of scheduling distributed data pipelines in cloud data platforms. The problem was formulated as a Markov decision process in which a controller jointly chooses node provisioning, the purchase market, and the dispatch rule, and it was solved with a double deep Q-network trained under domain randomization. In a simulator that includes deadline classes, bursty arrivals, provisioning delay, and a volatile spot market with price-dependent preemption, the controller reduced daily cost by 31.2 percent in the nominal scenario and by 33.6 to 52.6 percent in the other four scenarios relative to the strongest tuned reactive heuristic, while keeping SLA violations below 0.9 percent. The ablations showed that the spot-market action and the dispatch choice each contribute, with the former mainly affecting cost and the latter mainly protecting against overload.

The evaluation also exposed a significant weakness, since the learned policy changed cluster size far more often than the heuristics, and it is based entirely on simulation. These findings suggest a cautious route to adoption. Operators can use the approach to explore the cost and service trade-off explicitly, to prototype policies before they are applied to real clusters, and to run the controller in a shadow mode against production traces once the switching behavior has been regularized. With these safeguards, learned controllers are a promising component of cost-aware, adaptive data platforms.

Funding: Funding information was not supplied in the manuscript.

Conflicts of Interest: A conflict-of-interest declaration was not supplied in the manuscript.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers.

References

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., & Zaharia, M. (2010). A view of cloud computing. Communications of the ACM, 53(4), 50-58.
2. Chaisiri, S., Lee, B. S., & Niyato, D. (2012). Optimization of resource provisioning cost in cloud computing. IEEE Transactions on Services Computing, 5(2), 164-177.
3. Dean, J., & Barroso, L. A. (2013). The tail at scale. Communications of the ACM, 56(2), 74-80.
4. Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.
5. Delimitrou, C., & Kozyrakis, C. (2013). Paragon: QoS-aware scheduling for heterogeneous datacenters. Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, 77-88.
6. Delimitrou, C., & Kozyrakis, C. (2014). Quasar: Resource-efficient and QoS-aware cluster management. Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, 127-144.
7. Dulac-Arnold, G., Levine, N., Mankowitz, D. J., Li, J., Paduraru, C., Gowal, S., & Hester, T. (2021). Challenges of real-world reinforcement learning: Definitions, benchmarks and analysis. Machine Learning, 110(9), 2419-2468.
8. Dutreilh, X., Kirgizov, S., Melekhova, O., Malenfant, J., Rivierre, N., & Truck, I. (2011). Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow. Proceedings of the 7th International Conference on Autonomic and Autonomous Systems, 67-74.
9. Garcia, J., & Fernandez, F. (2015). A comprehensive survey on safe reinforcement learning. Journal of Machine Learning Research, 16, 1437-1480.
10. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., & Stoica, I. (2011). Dominant resource fairness: Fair allocation of multiple resource types. Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, 323-336.
11. Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., & Akella, A. (2014). Multi-resource packing for cluster schedulers. Proceedings of the ACM SIGCOMM Conference, 455-466.

12. Harlap, A., Tumanov, A., Chung, A., Ganger, G. R., & Gibbons, P. B. (2017). Proteus: Agile ML elasticity through tiered reliability in dynamic resource markets. Proceedings of the 12th European Conference on Computer Systems, 589-604.
13. Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). Deep reinforcement learning that matters. Proceedings of the 32nd AAAI Conference on Artificial Intelligence, 3207-3214.
14. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., & Stoica, I. (2011). Mesos: A platform for fine-grained resource sharing in the data center. Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, 295-308.
15. Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., & Goldberg, A. (2009). Quincy: Fair scheduling for distributed computing clusters. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, 261-276.
16. Jyothi, S. A., Curino, C., Menache, I., Narayanamurthy, S. M., Tumanov, A., Yaniv, J., Mavlyutov, R., Goiri, I., Krishnan, S., Kulkarni, J., & Rao, S. (2016). Morpheus: Towards automated SLOs for enterprise clusters. Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, 117-134.
17. Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. Proceedings of the 3rd International Conference on Learning Representations.
18. Lin, L. J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. Machine Learning, 8(3-4), 293-321.
19. Lorido-Botran, T., Miguel-Alonso, J., & Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. Journal of Grid Computing, 12(4), 559-592.
20. Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource management with deep reinforcement learning. Proceedings of the 15th ACM Workshop on Hot Topics in Networks, 50-56.
21. Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., & Alizadeh, M. (2019). Learning scheduling algorithms for data processing clusters. Proceedings of the ACM SIGCOMM Conference, 270-288.
22. Marcus, R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M., & Kraska, T. (2021). Bao: Making learned query optimization practical. Proceedings of the ACM SIGMOD International Conference on Management of Data, 1275-1288.
23. Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., & Tatbul, N. (2019). Neo: A learned query optimizer. Proceedings of the VLDB Endowment, 12(11), 1705-1718.
24. Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., & Dean, J. (2017). Device placement optimization with reinforcement learning. Proceedings of the 34th International Conference on Machine Learning, 2430-2439.
25. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.
26. Rao, J., Bu, X., Xu, C. Z., Wang, L., & Yin, G. (2009). VCONF: A reinforcement learning approach to virtual machines auto-configuration. Proceedings of the 6th International Conference on Autonomic Computing, 137-146.
27. Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., Strack, B., Witusowski, P., Hand, S., & Wilkes, J. (2020). Autopilot: Workload autoscaling at Google. Proceedings of the 15th European Conference on Computer Systems, 1-16.
28. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.
29. Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction (2nd ed.). MIT Press.
30. Tesauro, G., Jong, N. K., Das, R., & Bennani, M. N. (2006). A hybrid reinforcement learning approach to autonomic resource allocation. Proceedings of the IEEE International Conference on Autonomic Computing, 65-73.
31. Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., & Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 23-30.
32. van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double Q-learning. Proceedings of the 30th AAAI Conference on Artificial Intelligence, 2094-2100.
33. Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., & Baldeschwieler, E. (2013). Apache Hadoop YARN: Yet another resource negotiator. Proceedings of the 4th ACM Symposium on Cloud Computing, 1-16.
34. Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Large-scale cluster management at Google with Borg. Proceedings of the 10th European Conference on Computer Systems, 1-17.
35. Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., & Stoica, I. (2010). Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. Proceedings of the 5th European Conference on Computer Systems, 265-278.
36. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, 15-28.
37. Zhang, Q., Cheng, L., & Boutaba, R. (2010). Cloud computing: State-of-the-art and research challenges. Journal of Internet Services and Applications, 1(1), 7-18.