

A Dynamic Vertical Scaling Strategy for Distributed Stream Processing Applications in Edge Computing

Guilherme Hiago Costa dos Santos¹, Carlos Henrique Kayser¹, Tiago Ferreto¹

¹Polytechnic School, Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brazil

{guilherme.hiago, carlos.kayser}@edu.pucrs.br, tiago.ferreto@pucrs.br

Abstract. *Distributed Stream Processing applications at the edge must reconcile low latency and high throughput with limited and heterogeneous resources. This paper presents a dynamic vertical scaling strategy based on Proximal Policy Optimization, formulated as a Partially Observable Markov Decision Process. The policy jointly adjusts task allocations and prioritizes compliance with a service-level objective for the 95th-percentile end-to-end latency. In simulation-based experiments with two application profiles, two workloads, and ten paired placements per combination, the selected policies preserved throughput, achieved mean SLO violation rates ranging from 0.03% to 0.30%, with reductions of up to 8.55 percentage points compared with the baseline solution, and also used less CPU in three of four combinations. The comparison covers complete controller configurations with different decision frequencies.*

1. Introduction

The increasing volume and velocity of data generated by Internet of Things devices have exposed limitations of cloud-centric processing when applications require low response times or operate under constrained network bandwidth. Edge computing places computational resources closer to data sources, reducing unnecessary transfers and supporting time-sensitive processing outside centralized data centers [Shi et al. 2016, Shi and Dustdar 2016]. Edge infrastructures, however, offer less computational capacity than conventional clouds and comprise heterogeneous servers and network links. Resource management is therefore central to sustaining application performance at the edge.

Distributed Stream Processing (DSP) applications continuously process unbounded data streams through directed graphs of interconnected operators. Performance depends on the interaction among input rate, operator selectivity, processing demand, queue accumulation, network communication, and the resources assigned to each task. Distributed execution supports high throughput and low latency [Peng et al. 2019], but an allocation suitable for one workload can become inefficient as arrival rate or application pressure changes. Underprovisioned operators accumulate queues and increase end-to-end latency, whereas overprovisioning reserves scarce edge resources that could support other applications.

Dynamic scaling adapts resources at runtime through horizontal or vertical scaling. While horizontal scaling changes operator parallelism, vertical scaling modifies the resources of existing tasks, preserving the application graph and avoiding operator replication. Because operator demands and server capacities differ, uniform CPU changes may waste resources or preserve local bottlenecks. For example, VRebalance [Kang et al.

2021] uses Bayesian optimization to select vertical allocations from aggregate workload and performance feedback.

This work investigates whether operator-level observations allow a controller to allocate CPU selectively while maintaining application-level performance. Vertical scaling is formulated as a Partially Observable Markov Decision Process (POMDP) and controlled with Proximal Policy Optimization (PPO) [Schulman et al. 2017]. At one-second intervals, the policy selects relative CPU adjustments for every application task. Its reward prioritizes compliance with a 95th-percentile end-to-end latency Service Level Objective (SLO) and, while compliant, encourages lower CPU allocation and stable reconfiguration behavior. The research question is:

Can an operator-aware PPO policy balance CPU allocation efficiency with throughput preservation and p95 end-to-end latency SLO compliance in edge DSP applications?

The question is examined through three criteria: a lower mean run-level SLO violation rate than the selected baseline (H1); lower mean total CPU allocation without a higher mean SLO violation rate (H2); and profile-level throughput that follows the offered workload without a sustained deficit (H3). The contribution is the combination of task-pressure, host-capacity, and application-performance observations with joint incremental CPU actions and an SLO-aware reward in a POMDP formulation for fixed edge placements. PPO and vertical scaling are established techniques, the task-level representation and coordinated control design distinguish this implementation from application-level VRebalance. Their individual effects remain to be isolated experimentally.

The remainder of this paper is organized as follows. Section 2 reviews related work on dynamic scaling for edge stream processing. Section 3 defines the system model and vertical-scaling problem. Section 4 presents the operator-aware PPO autoscaler, including its observation space, action space, and reward. Section 5 describes the simulation environment, scenarios, baseline, and evaluation protocol. Section 6 reports and discusses the experimental results, and Section 7 concludes the paper and outlines future work.

2. Related Work

Existing proposals use heuristics, optimization, machine learning, or distributed control. The decentralized horizontal scaler in [Belkhiria and Tedeschi 2019] reacts to local and neighboring workloads. [Nardelli et al. 2018] coordinate operator, application, infrastructure, and region managers. Both change parallelism or active infrastructure rather than CPU assigned to fixed tasks.

[Peng et al. 2019] jointly optimize operator scaling and placement under QoS constraints. [Xu and Palanisamy 2021] use Markov decision processes and contextual bandits for operator parallelism, while [Arkian et al. 2021] use a performance model to add or remove geo-distributed resources. More recently, AgileDART [Ching et al. 2025] scales and places operators through a dynamic dataflow and adapts data-shuffling paths in heterogeneous edge networks. These systems address different scaling dimensions or execution settings.

VRebalance [Kang et al. 2021] is the closest baseline for this work because it modifies the CPU resources assigned to existing workers without changing their parallelism. It uses Bayesian optimization to explore task-allocation vectors and application-level feedback to balance 95th-percentile latency with resource cost. Its evaluation reported an

83.7% reduction in SLO violations compared with the default Apache Storm scaling algorithm. In contrast, the proposed policy observes individual task and host conditions and selects all relative task adjustments jointly. Table 1 compares all scaling approaches discussed above.

Table 1. Related studies compared by approach, scaling type, and target metric.

Work	Approach	Scaling type	Target metric
[Arkian et al. 2021]	Machine learning	Horizontal scaling	Throughput
[Xu and Palanisamy 2021]	Machine learning	Horizontal scaling	Throughput, latency
[Belkhiria and Tedeschi 2019]	Heuristic	Horizontal scaling	Throughput
[Peng et al. 2019]	Optimization (ILP)	Horizontal scaling	QoS compliance, cost
[Ching et al. 2025]	Heuristic	Horizontal scaling	Latency
[Nardelli et al. 2018]	Heuristic	Horizontal scaling	Latency, resources
[Kang et al. 2021]	Bayesian optimization	Vertical scaling	Latency, resources
This work	DRL (PPO)	Vertical scaling	Latency, resources, throughput

3. System Model and Problem Formulation

The infrastructure comprises heterogeneous edge servers $\mathcal{E}$ interconnected by network links. Data sources generate streams processed by a DSP application represented as a directed acyclic graph (DAG). Operators are materialized as tasks in $\mathcal{K}$. A server $\mathcal{E}_i$ has CPU capacity c_i , processing capacity $\hat{c}_i$, and RAM capacity m_i . Task k has computational demand h_k million instructions (MI) per event, memory demand d_k , and time-varying reserved CPU $c_k(t)$. Its placement $x_{i,k}$ remains fixed during an execution. When task k is placed on server i , its effective processing capacity in million instructions per second (MIPS) is

$$\sigma_{i,k}(t) = \hat{c}_i \frac{c_k(t)}{c_i}. \quad (1)$$

Increasing $c_k(t)$ increases the processing capacity assigned to the task and reduces its nominal per-event processing time. Its effect on application latency may be delayed because events can already be queued locally or at downstream operators.

End-to-end latency spans event generation at the source to completion at the sink, including queueing, network, and processing time. Let $L_{95}(t)$ denote the 95th percentile of the latency of events completed during interval t . The SLO is satisfied when $L_{95}(t) \leq g_j$, where g_j is the threshold for application profile j . Task input rate $\lambda_k(t)$, throughput $\phi_k(t)$, and inbound queue $q_k(t)$ represent offered work, processing progress, and accumulated pressure.

Each task receives between $c^{\min} = 500$ and $c^{\max} = 10000$ millicores. Allocations must obey

$$c^{\min} \leq c_k(t) \leq c^{\max}, \quad \sum_k x_{i,k} c_k(t) \leq c_i, \quad \forall i. \quad (2)$$

The only resource modified by the controller is the CPU allocation, while memory demand and placement are fixed. This choice follows VRebalance, which excludes mem-

ory from online resource configuration because low-latency stream processing should avoid costly storage operations on the critical processing path, while exploring memory configurations online may cause thrashing or other unpredictable behavior [Kang et al. 2021]. The control objective is hierarchical: first avoid or recover from SLO violations; then, while compliant, reduce unnecessary CPU without producing unstable reconfigurations or a sustained throughput deficit.

4. Operator-Aware Vertical Autoscaler

The autoscaler adjusts the CPU reserved for every task while preserving application topology and placement. At each interval, the agent receives a structured observation with task-, server-, and application-level information and selects a joint action containing one relative CPU adjustment per task. After the action is constrained by task and server capacities, The simulated environment advances the application for one simulated second and returns the next observation and reward. During training, transitions from multiple simulator instances update the PPO policy. During evaluation, the policy operates deterministically and its parameters remain fixed. Figure 1 summarizes this interaction.

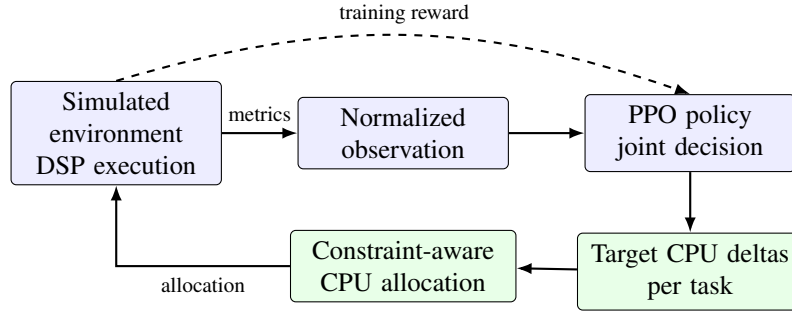


Figure 1. One-second observation, decision, and vertical-scaling control loop.

4.1. POMDP and Observation Space

The interaction is represented by the POMDP $\langle \mathcal{S}, \mathcal{A}, T, R, \Omega, O, \gamma \rangle$. The latent state contains the internal configuration required by the simulator dynamics, including workload progress, event queues and timestamps, network transfers, resource allocations, metrics, and temporal-history accumulators. PPO does not receive that state directly. A deterministic observation function constructs $o_t = O(s_t)$ from selected metrics, the policy samples or selects $\mathbf{a}_t$ from $\pi_\theta(\mathbf{a}_t | o_t)$, and a reward is produced after one second of simulated execution.

The observation is a dictionary of three flattened vectors,

$$o_t = \{\mathbf{o}_t^{\text{task}}, \mathbf{o}_t^{\text{server}}, \mathbf{o}_t^{\text{app}}\}. \quad (3)$$

There are 15 features per task, five per server, and 11 for the application. Table 2 groups these features by control purpose and normalization.

The representation combines global performance with localized evidence of pressure. The policy can distinguish an application-wide latency increase caused by one saturated operator from a condition in which all tasks have sufficient capacity. Because tasks

Table 2. Observation features, normalization, and control purpose.

Level	Feature group	Normalization	Control purpose
Task	CPU allocation; input rate; MI demand; throughput; inbound queue	CPU: clipped linear; rates, demand, and queue: $\log(1 + x)$	Represent capacity, offered work, processing demand, and accumulated pressure.
Task	Throughput/input ratio; queue pressure; demand/CPU	Ratios clipped to configured ranges; demand/CPU uses $\log(1 + x)$	Identify tasks whose assigned capacity is insufficient for their input.
Task	Task latency; processing latency; latency change; queue change	Latencies relative to g_j ; changes clipped to $[-2, 2]$	Expose local delay and whether pressure is increasing or decreasing.
Task	I/O ratio; host CPU utilization; host free CPU	I/O ratio and free CPU in $[0, 1]$; utilization clipped	Distinguish processing characteristics and capacity available for scale-up.
Server	CPU utilization/free CPU; memory utilization/free memory; task count	Capacity ratios clipped; task count normalized	Represent contention and feasibility of additional allocation on each host.
Application	Mean and p95 latency; SLO margin; p95 trend; violating-interval history	Latencies relative to g_j ; margin and trend clipped	Represent global compliance, proximity to violation, and recent behavior.
Application	Total, mean, maximum, and minimum task CPU; total queue; throughput	CPU linearly normalized; queue and throughput use $\log(1 + x)$	Summarize resource distribution and application-level work completion.

and servers occupy fixed positions in the flattened vectors, observation and action dimensions depend on the application graph. An independent policy is therefore trained for each profile.

4.2. Joint Action and Feasibility

The multi-discrete action contains one component per task:

$$\mathbf{a}_t = (a_{t,1}, \dots, a_{t,|\mathcal{K}|}), \quad a_{t,k} \in \{-500, -50, 0, 50, 500\}. \quad (4)$$

Each value is a relative CPU adjustment in millicores, applied to the current allocation and clipped to the task bounds. Coarse changes respond to large pressure variations, fine changes adjust allocation near an efficient point, and zero explicitly preserves a task allocation. Components are selected jointly, allowing the policy to increase a bottleneck task while reducing CPU elsewhere.

The target vector is projected onto the feasible set defined by Equation 2. Reductions are processed before increases so that capacity released on a host can be reused in the same decision. If a requested increase exceeds host free CPU, the applied vector differs from the target. The next observation and reward use the allocation actually applied, preventing credit for infeasible actions.

4.3. SLO-Aware Reward

The reward combines seven components and is clipped to $[-6, 2.5]$:

$$r_t = \text{clip}(R_{\text{SLO}} + R_{\text{risk}} + R_{\text{resource}} + R_{\text{progress}} + R_{\text{churn}} + R_{\text{recovery}} + R_{\text{reconf}}, -6, 2.5). \quad (5)$$

The safe-latency threshold is 80% of the profile SLO: 144 ms for PRED and 192 ms for ETL. While compliant, the agent receives 0.5 for meeting the SLO, a quadratic risk penalty of up to 0.5 as latency approaches the SLO, and a resource reward of up to 1.5 that increases as total applied CPU approaches the sum of task minima. CPU reduction contributes up to 0.25; returning from violation to compliance adds 0.5.

During violation, the resource, risk, and CPU-reduction terms are zero. The SLO penalty ranges from -1 to -5 and grows with relative violation severity. Recovery rewards reductions in p95 latency, and reconfiguration rewards scale-up aligned with task pressure, which combines throughput deficit, queue pressure, and processing latency. Churn penalizes allocation changes by at most 0.05 and is reduced during violation. These empirically selected coefficients encode an SLO-first preference, not a lexicographic guarantee; component effects require ablation.

Table 3. Reward components and their control roles.

Component	Range	Role
R_{SLO}	$[-5, 0.5]$	Rewards compliance and penalizes violations according to relative severity.
R_{risk}	$[-0.5, 0]$	Discourages operation close to the SLO while compliant.
R_{resource}	$[0, 1.5]$	Rewards low total CPU only while latency is compliant.
R_{progress}	$[-0.25, 0.25]$	Rewards CPU reduction and penalizes movement in the opposite direction during compliance.
R_{churn}	$[-0.05, 0]$	Penalizes unnecessary changes in task allocation.
R_{recovery}	$[-1, 1]$	Rewards lower latency during violation and transition back to compliance.
R_{reconf}	$[-0.5, 0.5]$	Rewards scale-up aligned with task pressure during SLO recovery.

5. Experimental Methodology

5.1. Simulation Environment and Scenarios

The evaluation uses EdgeStreamPy, a discrete-time simulator built on EdgeSimPy [Souza et al. 2023]. Switches route flows and edge servers provide CPU capacity and processing speed. A DSP application is a DAG of simulated tasks, each with MI demand per event and selectivity. The simulator models event generation, queues, processing, networking, and CPU reallocation; migration and horizontal scaling are disabled.

Each scenario has nine switches on a 3×3 hexagonal grid, 16 bidirectional switch links (1 Gbps, 10 ms), and seven edge servers attached to switches. The server models are Acer AR585 F1 (12,000 millicores, 26,400 MIPS, 32,768 MB RAM), SGI Rackable C2112-4G10 (16,000, 51,200, 32,768), and HPE ProLiant DL360 Gen9 (18,000, 41,400, 65,536). Tasks are randomly placed, at most two per server, and remain fixed. PRED and ETL are adapted from RIoT Bench [Shukla et al. 2017]; they contain six and nine simulated tasks, respectively, including their source and MQTT Publish sink operator. Business logic is represented by processing demand, selectivity, routing, and allocation rather than executed code.

Tables 4 and 5 list the per-task processing demand and selectivity used for reproducibility. Each source event is 2,000 bytes. Task memory demand is 2,048 MB except for SenMLParse (4,096 MB).

Table 4. PRED task parameters.

Task	MI/event	Selectivity
SOURCE	0.3	1
SenMLParse	0.73	2
LinearRegressionPredictor	0.3	1
DecisionTree	0.3	1
ErrorEstimation	0.1	0.5
MQTTPublish	0.25	1

Table 5. ETL task parameters.

Task	MI/event	Selectivity
SOURCE	0.3	1
SenMLParse	0.73	1
RangeFilter	0.0324	1
BloomFilter	0.0324	1
Interpolation	0.044	1
Join	1	1
Annotate	0.0324	1
CsvToSenML	0.22	1
MQTTPublish	0.25	1

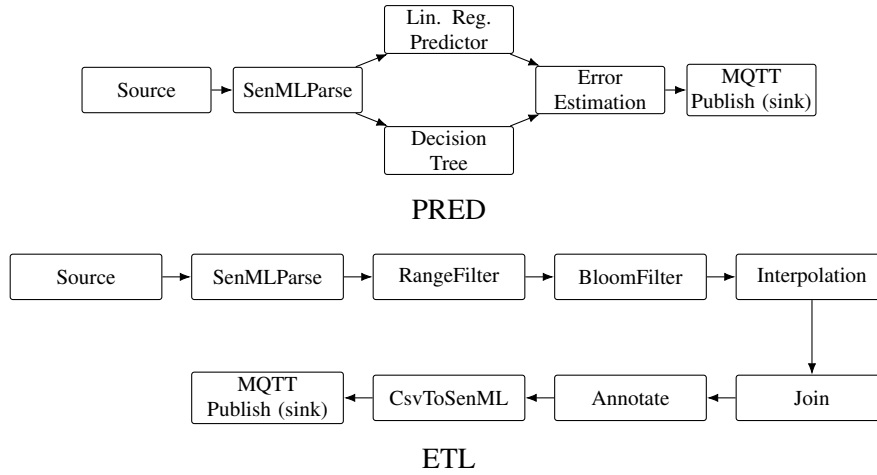


Figure 2. PRED and ETL application topologies adapted from RIOTBench.

Two arrival-rate traces are replayed. TAXI contributes the first 3,600 points of a RIOTBench smart-transportation trace beginning on January 1, 2010. INTEL derives from 54 sensors in the Intel Berkeley Research laboratory [Bodik et al. 2004]; its 1,056 points repeat cyclically over the 3,600-interval horizon. Source data are aggregated into five-minute windows and each point is replayed for one simulator second. Thus, each run has 3,600 one-second intervals (60 simulated minutes). INTEL cycles are repeated workload observations, not independent replications.

The generic SLO is instantiated separately for each profile: 240 ms for ETL and 180 ms for PRED. ETL uses a higher latency reference because its operator chain is longer than PRED’s. The same profile-specific thresholds are applied to both controllers under both workloads. The SLO violation rate is the percentage of one-second intervals in which p95 end-to-end latency exceeds the profile threshold. Throughput is the number of events reaching the sink during the interval, and allocation is the sum of CPU reserved for all profile tasks.

5.2. Baseline, Training, and Protocol

VRebalance was reimplemented for EdgeStreamPy. Its Bayesian optimization procedure scores an allocation from application p95 latency and allocated CPU, while sink throughput selects a workload tier. Each tier retains an optimizer and observation history, with a

5–16-minute exploration window. All tasks start at 500 millicores, the first allocation is held for 60 simulator steps and subsequent updates occur every 60 steps. A one-second VRebalance control interval was also tried in preliminary tuning, but the 60-second configuration performed better and was retained; paired results for the one-second variant are not reported here. Candidate allocations use PPO’s 500–10,000 millicore bounds, quantized in 50-millicore increments. The bounds were altered from the original VRebalance to fit this infrastructure. When a request exceeds host capacity, each task retains its minimum and the remaining CPU is distributed proportionally to requested demand above that minimum.

Independent PPO models were trained for PRED and ETL because their observation and action dimensions differ. Training used a 352-point segment from the penultimate taxi day, separate from the final TAXI trace, it retained workload increases and decreases while limiting training cost. Original and doubled arrival rates were combined with placements from seeds 42 and 37, giving four scenarios per profile. These variations address poor transfer observed in preliminary single-placement training. Each model had a 500,000-step budget over 14 parallel simulator instances and stopped after 516,096 transitions. Stable-Baselines3 PPO used `MultiInputPolicy` with separate actor and critic networks of two 64-unit layers, Table 6 gives the remaining settings. One policy per profile was selected by mean development reward. The recorded training elapsed time was approximately three hours per model on an NVIDIA GeForce RTX 4060 Ti (8 GB), with Python 3.9.13 and PyTorch 2.8.0/CUDA 12.9.

Table 6. PPO training configuration used for both application profiles.

Learning rate	3×10^{-4}	Steps/rollout/environment	2,048
Minibatch size	256	Optimization epochs	20
Discount factor γ	0.995	GAE factor λ	0.95
Policy clipping	0.20	Value-function clipping	0.20
Entropy coefficient	0.05	Value-function coefficient	0.50
Maximum gradient norm	0.50	Training seed	284572

The two profiles and workloads yield four combinations. Ten random placements per combination and controller give 80 runs, each PPO run is paired with VRebalance using the same workload, placement, and evaluation seed. Scenario and application seeds were randomly drawn, with distinct seeds for each combination of application and workload. Evaluation placements differ from training placements. PPO is deterministic and reuses one policy per profile, so the ten repetitions measure placement, not independent training variability.

The comparison covers complete configurations: PPO acts every second and VRebalance every 60 seconds. For H1, paired differences in run-level SLO violation rates receive 95% percentile-bootstrap confidence intervals; paired Wilcoxon tests compare the ten placements, with Holm correction across four combinations. For H3, each run’s throughput is the mean over 3,600 intervals. Each matched pair yields the relative difference $(\bar{\phi}_{\text{PPO}} - \bar{\phi}_{\text{VRebalance}}) / \bar{\phi}_{\text{VRebalance}} \times 100$. A 95% percentile-bootstrap interval for the mean paired difference uses 10,000 resamples of complete pairs. PPO is considered non-inferior when its lower bound exceeds -5% .

6. Experimental Results

6.1. Tail Latency and SLO Mitigation

Across the repeated paired executions, PPO shows more consistent p95 latency compliance than VRebalance in all four combinations. The following analyses distinguish temporal behavior, run-level means, and SLO violation rates.

Figure 3 shows the temporal behavior behind these run-level summaries. The pointwise mean p95 latency of PPO remains below the corresponding SLO throughout execution in all four combinations. VRebalance presents repeated excursions above the thresholds, with more pronounced variation among placements for the PRED profile.

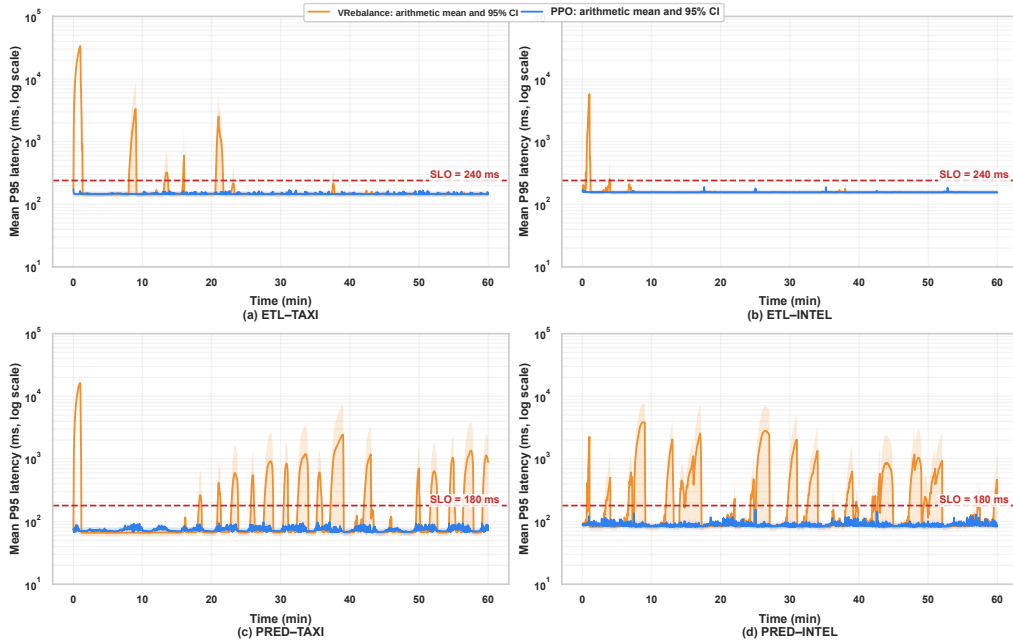


Figure 3. p95 end-to-end latency over time. Lines are pointwise arithmetic means for the ten executions.

The paired run-level means in Figure 4 show that this group pattern is not solely a consequence of averaging compliant and non-compliant executions. Every PPO execution has a mean p95 latency below its profile-specific SLO. For VRebalance, ETL-INTEL is the only scenario in which every run-level mean remains below the threshold. ETL-TAXI presents higher run means despite a lower violation rate than the PRED scenarios because a small number of extreme intervals strongly affect the mean.

Figure 5 reports run-level violations. PPO’s mean rates are 0.30%, 0.03%, 0.10%, and 0.18% for ETL-TAXI, ETL-INTEL, PRED-TAXI, and PRED-INTEL; VRebalance’s are 2.87%, 1.01%, 5.72%, and 8.73%. Paired mean reductions are 2.57, 0.98, 5.61, and 8.55 percentage points, with respective 95% bootstrap intervals of [1.58, 3.46], [0.73, 1.31], [1.71, 11.15], and [1.52, 16.95]. Wilcoxon tests with Holm correction give adjusted $p = 0.00781$ in each combination. Thus H1 is supported for the complete configurations tested, with uncertainty computed across placements. PRED-INTEL also includes base-line runs above 30% for VRebalance, revealing sensitivity hidden by its mean.

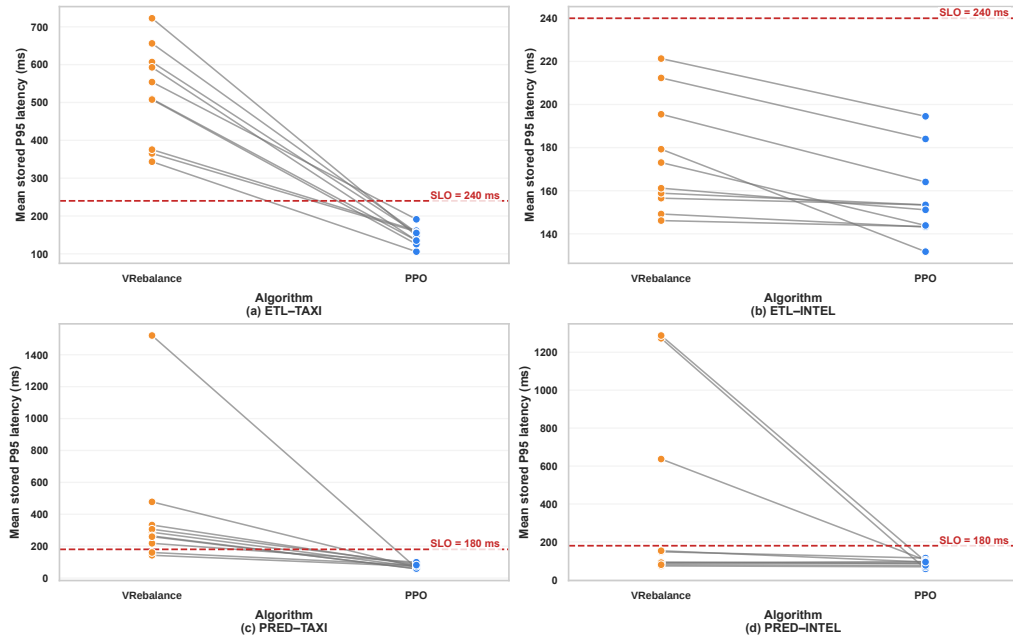


Figure 4. Paired per-execution mean p95 end-to-end latency. Dashed horizontal lines are the profile-specific SLOs.

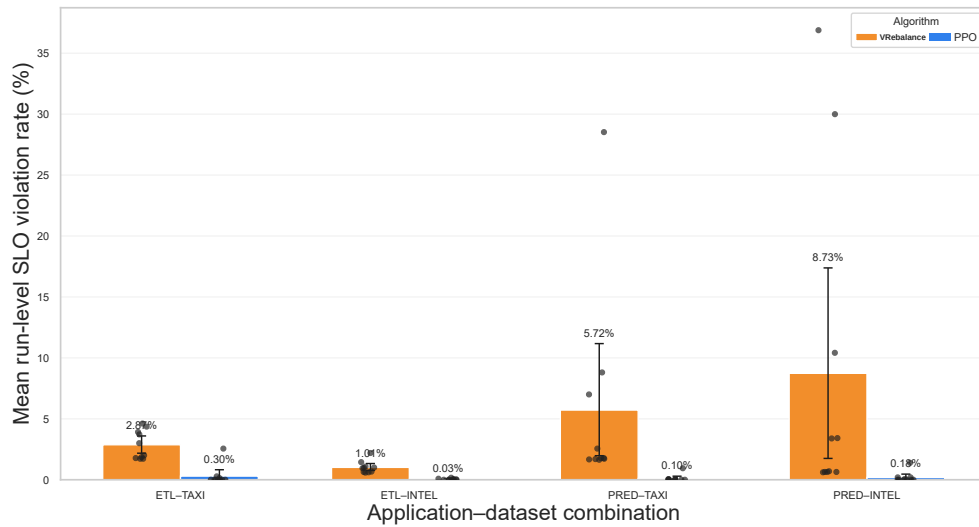


Figure 5. Run-level p95 latency SLO violation rates. Bars are means of ten run-level rates, points are executions, and whiskers are 95% percentile confidence intervals.

6.2. CPU Allocation Efficiency Under Compliance

H2 concerns CPU efficiency while preserving the SLO behavior established under H1, rather than minimum allocation in isolation. Figure 6 shows lower CPU under PPO in three scenarios. For ETL–TAXI, ETL–INTEL, and PRED–TAXI, mean and median allocations are lower, with central-tendency differences of approximately 15,000–25,000 millicores. PPO allocation remains below VRebalance for more than 80% of the horizon in these scenarios and follows workload changes at its one-second control interval.

PRED–INTEL is the exception: PPO’s median allocation is slightly higher. In that scenario, however, PPO has a mean violation rate of 0.18%, compared with 8.73% for VRebalance, the baseline’s highest rate among the combinations. This joint result is consistent with the reward hierarchy: the policy accepts more CPU when the simulated profile is under greater SLO pressure. It does not demonstrate that additional CPU alone caused the lower violation rate. H2 is therefore not supported uniformly; the evidence indicates a scenario-dependent tendency toward resource savings.

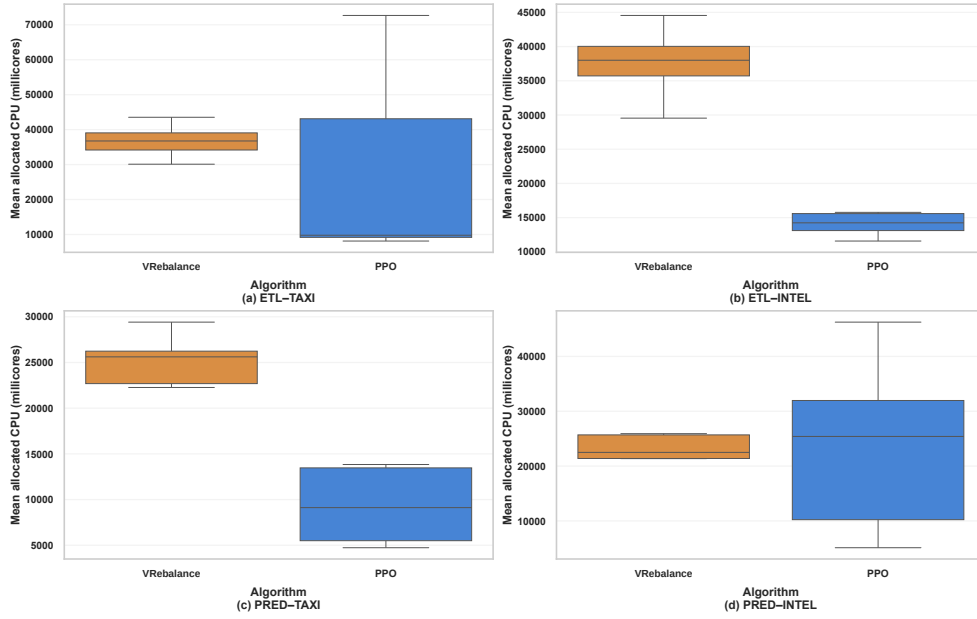


Figure 6. Distribution of per-execution mean total CPU allocation. The center line is the median.

6.3. Throughput Preservation

Figure 7 compares offered workload and achieved throughput. After the initial transient, the median controller curves closely follow the offered-rate trajectory. The paired non-inferiority test used a -5% margin, and PPO’s throughput remained within the predefined $\pm 5\%$ band relative to VRebalance in all four combinations.

The mean paired differences (PPO minus baseline) are approximately 0.001% for ETL–TAXI and ETL–INTEL, 0.026% for PRED–TAXI (95% CI: -0.005% to 0.069%), and -0.001% for PRED–INTEL (95% CI: -0.003% to 0.001%). All lower confidence bounds exceed the -5% margin, supporting H3 in every combination.

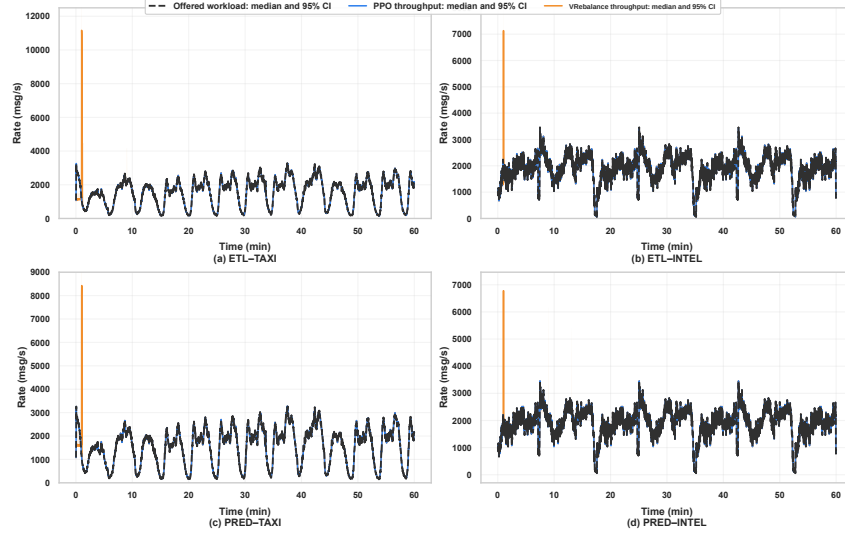


Figure 7. Offered workload and achieved throughput for the four profile–workload combinations.

7. Conclusion and Future Work

This work investigated whether an operator-aware DRL controller can reduce CPU allocation for edge DSP applications while preserving throughput and p95 end-to-end latency SLO compliance relative to VRebalance. Across the four profile–workload combinations, PPO achieved mean SLO violation rates of 0.03–0.30%, versus 1.01–8.73% for VRebalance. Paired confidence intervals and Holm-adjusted tests support H1, and throughput met the non-inferiority criterion in every combination (H3). PPO used less CPU in three combinations, in PRED–INTEL, it used slightly more but reduced mean violations from 8.73% to 0.18%. H2 therefore has mixed support.

The study compares selected complete controllers, not isolated learning algorithms: PPO updates every second and VRebalance every 60 seconds. Preliminary tuning of VRebalance at one-second intervals does not replace a paired frequency ablation, so the observed benefit cannot be assigned specifically to PPO, operator-aware observations, or update frequency. Ten placements share a single trained policy per profile. The fixed-placement simulation, two workloads, and absent inference/actuation overhead measurements further limit generalization to deployed edge engines.

Future work should test independent training seeds, additional profiles, workloads, and topologies; ablate observation groups, reward terms, action increments, and decision frequency; and evaluate a stream-processing engine on emulated and physical edge infrastructure, measuring inference, metric-collection, and actuation overheads.

Acknowledgments

This work was supported by the PDTI Program, funded by Dell Computadores do Brasil Ltda (Law 8.248/91). The authors acknowledge the High-Performance Computing Laboratory of the Pontifical Catholic University of Rio Grande do Sul for providing resources for this project.

References

- Arkian, H., Pierre, G., Tordsson, J., and Elmroth, E. (2021). Model-based stream processing auto-scaling in geo-distributed environments. In *2021 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–10. IEEE.
- Belkhiria, M. M. and Tedeschi, C. (2019). Design and evaluation of decentralized scaling mechanisms for stream processing. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 247–254. IEEE.
- Bodik, P., Hong, W., Guestrin, C., Madden, S., Paskin, M., and Thibaux, R. (2004). Intel lab data. <https://db.csail.mit.edu/labdata/labdata.html>. Data collected from 54 sensors deployed in the Intel Berkeley Research lab.
- Ching, C.-W., Chen, X., Kim, C., Wang, T., Chen, D., Da Silva, D., and Hu, L. (2025). AgileDART: An agile and scalable edge stream processing engine. *IEEE Transactions on Mobile Computing*, 24(5):4510–4528.
- Kang, P., Lama, P., and Khan, S. U. (2021). SLO-aware virtual rebalancing for edge stream processing. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 126–135. IEEE.
- Nardelli, M., Russo Russo, G., Cardellini, V., and Lo Presti, F. (2018). A multi-level elasticity framework for distributed data stream processing. In *European Conference on Parallel Processing*, pages 53–64. Springer.
- Peng, Q., Xia, Y., Wang, Y., Wu, C., Luo, X., and Lee, J. (2019). Joint operator scaling and placement for distributed stream processing applications in edge computing. In *Service-Oriented Computing: 17th International Conference, ICSOC 2019*, pages 461–476. Springer.
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.
- Shi, W., Cao, J., Zhang, Q., Li, Y., and Xu, L. (2016). Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646.
- Shi, W. and Dustdar, S. (2016). The promise of edge computing. *Computer*, 49(5):78–81.
- Shukla, A., Chaturvedi, S., and Simmhan, Y. (2017). RIOTBench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257.
- Souza, P. S., Ferreto, T., and Calheiros, R. N. (2023). EdgeSimPy: Python-based modeling and simulation of edge computing resource management policies. *Future Generation Computer Systems*, 148:446–459.
- Xu, J. and Palanisamy, B. (2021). Model-based reinforcement learning for elastic stream processing in edge computing. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 292–301. IEEE.