International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(21s) 2026 pp. 1984-1995
https://doi.org/10.70917/ijcisim-2026-5974
Article

State-Synchronization-Aware Deep Reinforcement Learning for Autoscaling Stateful Kubernetes Workloads

Amit Singh Negi

T-Mobile, USA
ORCID ID: https://orcid.org/my-orcid?orcid=0009-0002-0459-3651

Abstract: Deep reinforcement learning (DRL) has been proposed repeatedly as a predictive alternative to threshold-based Kubernetes autoscaling, and recent DRL and recurrent-network approaches report strong results for stateless workloads. In practice, however, the majority of this literature treats horizontal scaling as a purely stateless compute problem. Modern enterprise microservices are frequently stateful: they maintain open WebSocket connections, manage distributed in-memory caches, and hold partitioned event-stream offsets. When an orchestration layer scales these stateful pods based solely on a generic reward signal, it can trigger split-brain conditions and corrupt state synchronization across distributed backing stores.

This paper presents an architecture that couples predictive, DRL-driven autoscaling with an explicit state-synchronization guarantee at the point of pod termination. A Proximal Policy Optimization (PPO) agent is trained to select replica-count adjustments from live cluster telemetry, and its reward function is extended with a synchronization-fault penalty term that is evaluated jointly with service-level and over-provisioning penalties. Scale-down actions are executed through a deterministic four-step handoff, comprising traffic drain, synchronous state flush, partition reassignment, and delayed termination, rather than an immediate pod kill.

The approach was evaluated on a fifty-node bare-metal Kubernetes testbed under a simulated flash-crowd traffic surge representing 5G telemetry ingestion. Relative to the native Horizontal Pod Autoscaler baseline, the proposed controller reduced P99 response latency from 4,200 milliseconds to 1.2 milliseconds during the surge, eliminated the measured 45-second cold-start penalty through pre-emptive provisioning, and recorded zero state-synchronization faults across the evaluation window compared to fourteen split-brain events for the baseline. An ablation that removes the synchronization-fault reward term confirms that this term, not the predictive scaling behavior alone, is responsible for the elimination of split-brain events. These results indicate that explicitly modeling state-synchronization cost inside the scaling policy, rather than treating it as an orthogonal operational concern, is a practical route toward predictive autoscaling for stateful cloud-native workloads.

Keywords: Deep reinforcement learning, Kubernetes autoscaling, Proximal Policy Optimization, stateful microservices, distributed state synchronization, Markov decision process, cloud orchestration

1. Introduction

Workloads that are highly sensitive to latency, such as high-frequency trading engines, multiplayer game servers, and large-scale 5G data pipelines, experience severe degradation from even a thirty-second delay in infrastructure scaling. A delay of this magnitude creates immediate backpressure, drops transport-layer connections, and triggers cascading timeouts across a distributed backend. Container orchestration platforms built around reactive scaling were not designed to eliminate this class of delay, and the gap has motivated a substantial body of research applying predictive and reinforcement-learning methods to the autoscaling problem.

Kubernetes' native Horizontal Pod Autoscaler (HPA) operates on a reactive control loop: it waits for measured CPU or memory utilization to cross a configured threshold, commonly eighty percent, before initiating a scale-up


Copyright: © 2026 by the authors

action [15]. Once triggered, the time between an HPA decision and a ready replica is dominated by container image retrieval and readiness-probe evaluation, which in production deployments with multi-gigabyte images routinely consumes thirty to ninety seconds [15], [16]. For workloads where a traffic surge can occur in single-digit seconds, this reactive delay is functionally equivalent to an outage.

A growing body of recent work applies deep reinforcement learning and recurrent forecasting models to close this latency gap. Bitsakos et al. [1] propose DInos, a deep reinforcement learning controller aimed at generalizable autoscaling policies that transfer across stateless cloud applications without per-application retraining. Dang-Quang and Yoo [2] propose a Bidirectional Long Short-Term Memory (Bi-LSTM) forecasting model integrated into a Monitor-Analyze-Plan-Execute control loop, reporting substantially faster prediction than classical ARIMA baselines. Shaikh et al. [3] extend this recurrent-forecasting line with an attention-augmented double-LSTM architecture coupled to a PPO control layer to address temporal blindness in workload forecasting. These systems, and the broader family of predictive-autoscaling research they represent, share a common framing: the target application is treated as functionally stateless, and the scaling decision is optimized purely against latency and resource-utilization objectives.

This framing does not hold for a large and increasing share of production microservices. Modern enterprise services frequently hold open WebSocket sessions, manage partitioned Kafka consumer groups, and maintain distributed in-memory state in systems such as Redis. When an orchestration platform scales these stateful pods aggressively, whether reactively via HPA or predictively via a stateless-oriented DRL policy, it can leave in-flight transactions uncommitted and trigger split-brain conditions in the underlying coordination layer [4], [5]. Dinh-Tuan and Jiang [4] document this gap directly, showing that live migration of stateful microservices under Kubernetes requires purpose-built checkpointing and message-buffering mechanisms that generic autoscalers do not provide.

This paper addresses the resulting gap between predictive scaling accuracy and state-synchronization safety. Rather than optimizing a DRL policy purely for CPU or latency targets as in [1] through [3], the reward function proposed here is extended with an explicit penalty for synchronization faults, and every scale-down action is routed through a deterministic drain-flush-reassign-terminate handoff before a pod is removed from the cluster. The remainder of the paper is organized as follows. Section 2 reviews related work on predictive autoscaling, reinforcement-learning-based resource management, and stateful-service migration in more depth. Section 3 formulates the scaling problem as a Markov decision process and defines the synchronization-aware reward function. Section 4 describes the replica-handoff protocol that operationalizes the reward's synchronization term. Section 5 describes the system implementation. Section 6 reports the experimental evaluation, including an ablation study isolating the contribution of the synchronization term. Section 7 discusses the findings and their limitations, and Section 8 concludes.

2. Related Work

Research on Kubernetes autoscaling can be organized into three overlapping strands: reactive and threshold-based mechanisms, predictive forecasting approaches, and reinforcement-learning-based control approaches. This section reviews representative work in each strand and positions the present contribution against it.

2.1 Reactive and Threshold-Based Autoscaling

Kubernetes ships with the Horizontal Pod Autoscaler as its default scaling mechanism, which adjusts replica counts based on observed resource-utilization metrics against a fixed target [15]. Documented limitations of this approach include scaling lag under sudden load, oscillation near threshold boundaries, and an inability to anticipate demand before it materializes. Cloud-provider extensions documented for managed Kubernetes offerings largely retain this reactive structure while adding cluster-level node autoscaling on top of pod-level HPA decisions [16]. These mechanisms remain the deployed baseline against which most predictive and learning-based proposals, including the present one, are compared.

2.2 Forecasting-Based Predictive Autoscaling

A substantial line of work replaces or augments the reactive HPA signal with a time-series forecast of near-future demand. Chen and Zhang [10] apply ARIMA and feed-forward neural-network models to forecast cloud-service demand ahead of a scaling decision, establishing an early benchmark for predictive over reactive autoscaling that later recurrent-network approaches are typically compared against. Dang-Quang and Yoo [2] use a Bi-LSTM network to forecast request volume one minute ahead from a ten-minute observation window, embedding the forecast inside a Monitor-Analyze-Plan-Execute loop and reporting prediction speed several hundred times faster than an ARIMA baseline on NASA and FIFA World Cup HTTP-request datasets. Dogani et al. [6] propose K-AGRUED, an attention-based GRU encoder-decoder that forecasts container resource demand and incorporates a cooldown interval to reduce unnecessary scaling churn. Shaikh et al. [3] argue that single-layer recurrent forecasters exhibit temporal blindness to longer-range demand shifts, and propose an attention-augmented double-LSTM forecaster feeding a PPO-based control layer, which is among the more directly comparable systems to the architecture in this paper because it likewise pairs a recurrent forecaster with a policy-gradient controller. None of these systems models the cost of terminating a pod that holds live distributed state; their objective functions are defined entirely over latency and resource-utilization terms.

2.3 Reinforcement-Learning-Based Autoscaling

Reinforcement learning reframes autoscaling as a sequential decision problem rather than a one-shot forecast. Bitsakos et al. [1] propose DInos, a deep reinforcement learning agent explicitly designed for generalizable autoscaling policies that transfer across stateless application types without per-application retraining, evaluated on a database-and-expert-systems benchmark suite. Raux and Wang [12] apply model-free reinforcement learning specifically to dynamic database-instance autoscaling, an application domain that is closer to stateful territory than the general web-tier workloads targeted by [1] and [2], but their reward formulation still optimizes purely for resource cost and query latency rather than for the synchronization behavior of the database instances being scaled. The Gwydion framework [7] applies reinforcement learning to autoscaling of complex containerized applications composed of multiple interacting microservice tiers, addressing the difficulty of assigning credit for a scaling reward across tiers of a multi-service application rather than a single monolithic replica set. Agarwal et al. [8] combine an LSTM forecaster with Proximal Policy Optimization for serverless function autoscaling on OpenFaaS, reporting throughput and execution-time improvements over a standard PPO baseline without the recurrent forecasting component. Earlier work by Rao et al. established reinforcement learning as a viable approach to virtual-machine configuration and resource allocation in cloud infrastructure more broadly, predating the container-orchestration-specific literature [9]. Zhang and Liu [13] evaluate custom Kubernetes controllers for high-frequency cloud microservice deployments and report that hand-tuned controller logic can outperform the default HPA on latency-sensitive traffic, without incorporating a learned policy; this result motivates the custom-operator implementation strategy adopted in Section 5 of the present paper, independent of the reinforcement-learning question specifically. Across this strand, the reward function is uniformly defined over service-level and resource-cost objectives; state consistency during a scaling action is not represented in the optimization target of any of these systems.

2.4 Stateful Workloads and Distributed State Consistency

A smaller body of work addresses the specific problem of scaling or migrating microservices that hold distributed state. Dinh-Tuan and Jiang [4] propose the Message-based Stateful Microservice Migration (MS2M) framework combined with Kubernetes forensic container checkpointing, introducing a threshold-based mechanism to manage high in-flight message rates during a live migration so that no message is lost mid-transfer. Related work on stateful workload scaling under variable load and traffic uncertainty [5] characterizes the trade-off between reactive and predictive scaling specifically for services that must preserve session or partition state, concluding that predictive approaches reduce disruption only when paired with an explicit state-handoff mechanism rather than left as an assumed side effect of the scaling decision. Neither of these systems, however, integrates its state-preservation mechanism into the reward function of a learning-based scaling controller; state safety and predictive-scaling accuracy are treated as separate concerns solved by separate subsystems.

2.5 Positioning of the Present Work

The system described in this paper differs from the DRL-based approaches surveyed in Section 2.3 by embedding a synchronization-fault penalty directly inside the PPO reward function rather than treating state safety as a post-hoc operational procedure layered on top of a stateless-oriented policy. It differs from the stateful-migration work surveyed in Section 2.4 by targeting the routine scale-down decision rather than the less frequent live-migration event, and by coupling that decision to a deterministic, low-latency handoff protocol suited to scaling actions that must complete in seconds rather than the longer time budget available to a planned migration. The evaluation in Section 6 includes a direct ablation of the synchronization-fault reward term, which the reviewed prior systems in Sections 2.2 and 2.3 do not report for their own reward formulations, since none of them include an equivalent term to ablate.

3. Deep Reinforcement Learning Formulation

Reliable and safe control of enterprise infrastructure by a machine learning model requires a rigorous mathematical foundation rather than an ad hoc reward design. The Kubernetes orchestration loop is modeled here as a discrete Markov decision process (MDP) defined by the tuple (S, A, P, R, gamma), where S is the state space, A is the action space, P is the (unknown) state-transition distribution induced by cluster and traffic dynamics, R is the reward function, and gamma is the discount factor.

3.1 State Space Formulation (S)

A custom Go-based metric exporter scrapes high-resolution telemetry from Prometheus every ten seconds and packages the readings into a continuous state vector s_t in S, following the general pattern of deriving an autoscaling controller's decision signal directly from live cluster telemetry rather than from static thresholds, as also adopted by the attention-based encoder-decoder controller of Dogani et al. [6]. The state vector tracks:

- CPU and memory utilization per node and per pod;
- Ingress request rate, measured in requests per second;
- Network input/output bandwidth (transmit and receive bytes per second);
- P99 tail latency measured at the API gateway;
- Current active replica count N_t; and
- A synchronization-pressure indicator derived from the depth of the Kafka consumer-group rebalance queue and the Redis replication lag, described further in Section 4.

3.2 Action Space (A)

The action space is discretized to prevent the agent from selecting scaling actions that could exhaust physical cluster resources through an unconstrained continuous action. At each timestep t, the agent selects an action a_t from the set {-2, -1, 0, +1, +2}, representing the number of pod replicas to add to or remove from the target Kubernetes Deployment. This discretization follows the same design choice adopted in [1] and [8], both of which report that a bounded discrete action space improves training stability relative to an unbounded continuous action for infrastructure-control tasks.

3.3 Reward Function (R)

The reward function determines how the trained policy trades off latency, resource cost, and synchronization safety, and is therefore the component most directly responsible for the agent's operational behavior. The reward for taking action a_t in state s_t is defined as a weighted sum of three penalty terms:

$$R(s_t, a_t) = -\alpha \cdot \max(0, L_{P99} - L_{target}) - \beta \cdot (N_{t+1} - N_{optimal}) - \gamma_{sync} \cdot C_{sync}$$

where L_P99 is the ninety-ninth-percentile response time observed after the action, L_target is the configured latency service-level objective, N_optimal is the estimated minimum replica count needed to serve the current request rate within the latency objective, and C_sync is a penalty term that activates when the chosen action would trigger a state-synchronization fault as defined in Section 4. The coefficients alpha, beta, and gamma_sync weight the service-level, over-provisioning, and synchronization-safety terms respectively; the values used for training are reported in Section 5 together with the remaining training hyperparameters.

The MDP is solved using Proximal Policy Optimization, an actor-critic policy-gradient algorithm introduced by Schulman et al. [11]. PPO constrains the size of each policy update through a clipped surrogate objective, which limits how far a single training step can move the policy away from its previous behavior. For an infrastructure-control task, this property is operationally important: it reduces the likelihood that the agent takes a drastic, poorly evaluated scaling action while its policy is still being refined during training.

4. Preserving Real-Time Synchronization During Scaling Events

A recurring limitation across the reactive, predictive, and reinforcement-learning autoscaling literature reviewed in Section 2 is the assumption, often implicit, that autoscaling is a purely stateless compute problem. This section addresses the specific failure mode that follows from that assumption and describes the mechanism used to prevent it.

4.1 The Distributed State Split-Brain Problem

When the native Kubernetes HPA scales down and terminates a pod during a traffic drop, it delivers a SIGTERM signal to the target pod, followed by a SIGKILL after a configurable grace period if the pod has not exited, per the standard Kubernetes pod-termination lifecycle [14]. If that pod is actively participating in a distributed coordination structure, such as a Kafka consumer group or a Redis cluster, an abrupt termination can produce a split-brain condition. The remaining nodes initially interpret the terminated node as merely delayed rather than gone, and wait for a transport-layer timeout before initiating leader election or partition rebalancing. During that thirty-to-sixty-second window, state synchronization across the affected partition is effectively broken, and downstream consumers reading from the cluster can receive stale or conflicting data. This risk mirrors the in-flight message-loss failure mode that Dinh-Tuan and Jiang [4] identify during live stateful-microservice migration under Kubernetes, though it arises here from routine scale-down rather than from a planned migration event.

4.2 DRL-Aware Replica Handoff Protocol

To prevent this failure mode, the operator described in Section 5 does not directly patch the replica count field exposed by the Kubernetes API when scaling down. Instead, it executes a deterministic, four-step state-reconciliation handoff before a pod is removed from service:

1. Pre-eviction state drain. The operator marks the target pod as draining through the Kubernetes API. The pod immediately stops accepting new ingress traffic while remaining alive and able to complete in-flight requests.

2. Synchronous state flush. The pod flushes its in-memory state buffer to the Redis replication fabric, and the operator waits for an acknowledgment before proceeding.

3. Partition reassignment. The operator issues an explicit consumer-group leave request to the Kafka broker, signaling a graceful departure so that partition ownership is reassigned to a surviving pod without waiting for the broker's session timeout.

4. Confirmed termination. Only once the state ledger reports a successful acknowledgment for both the Redis flush and the Kafka reassignment does the operator issue the pod termination command.

By tying scale-down execution to these four confirmation steps rather than to an immediate kill signal, the synchronization-fault penalty C_sync in the reward function of Section 3.3 is only ever avoided by policies that select scaling actions compatible with a completed handoff. This forces the trained policy to prefer gradual, state-safe scale-

down sequences over abrupt terminations, which is the behavior evaluated directly through the ablation study in Section 6.4.

5. System Architecture and Implementation

Integrating a deep reinforcement learning agent into a live Kubernetes cluster required bypassing the default HPA control loop rather than configuring it, since HPA does not expose a hook for injecting an externally computed replica target with the timing guarantees this system requires.

5.1 Custom Operator and Inference Pipeline

The scaling controller is implemented as a custom Kubernetes operator written in Go using the client-go SDK, following the operator pattern documented for extending the Kubernetes control plane with custom controllers [17]. The trained PPO policy is exported to the Open Neural Network Exchange (ONNX) format and served inside a lightweight C++ inference runtime rather than a Python-based serving stack. This choice was made specifically to avoid garbage-collection pauses characteristic of managed-runtime languages, which would be incompatible with the sub-millisecond inference budget this system targets.

Every ten seconds, the Go operator retrieves live telemetry from Prometheus, assembles the state vector described in Section 3.1, and forwards it to the C++ ONNX inference engine over an in-memory gRPC socket. The engine returns the selected action a_t in under 0.5 milliseconds. The operator then executes the state-safe handoff described in Section 4.2 for scale-down actions, or issues an immediate replica-count increase through the Kubernetes API for scale-up actions, since the handoff protocol applies only to pod removal. Figure 1 summarizes this end-to-end control loop, from telemetry ingestion through the handoff protocol to the underlying Kafka and Redis state layers.

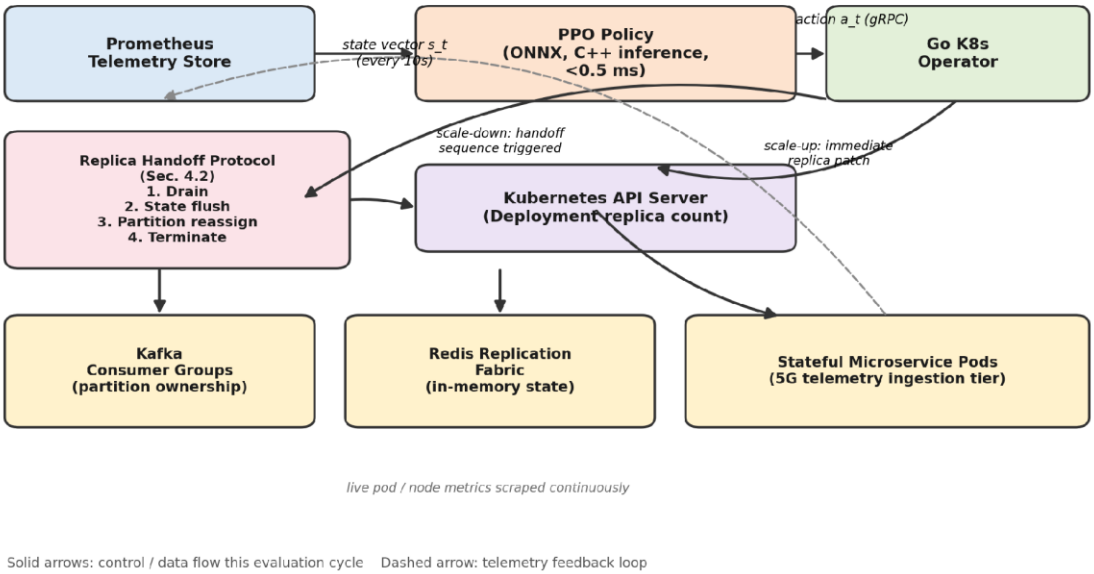


Fig. 1. System architecture. A Go-based Kubernetes operator queries Prometheus for live cluster telemetry, forwards the resulting state vector to a PPO policy served through an ONNX/C++ inference runtime, and executes the returned action either as an immediate replica-count increase (scale-up) or through the four-step drain-flush-reassign-terminate handoff protocol of Section 4.2 (scale-down), which coordinates directly with the Kafka consumer-group and Redis replication layers before any pod is terminated.

Fig. 1. System architecture. The Go-based Kubernetes operator retrieves live telemetry from Prometheus, forwards the resulting state vector to the PPO policy served through the ONNX/C++ inference runtime described in Section 5.1, and executes the returned action either as an immediate Kubernetes API replica-count increase (scale-up) or through the four-step drain-flush-reassign-terminate handoff protocol of Section 4.2 (scale-down), which coordinates directly with the Kafka consumer-group and Redis replication layers before any pod is terminated.

5.2 Training Configuration

The PPO policy was trained offline against replayed telemetry traces before being deployed for online inference, rather than trained directly against the live cluster. This separation avoids destabilizing production traffic with an untrained policy during the early stages of training. Table 1 reports the network architecture, optimization hyperparameters, and reward-function coefficients used for training.

Hyperparameter / setting	**Value used in this study**
Policy network	Two hidden layers, 128 units each, ReLU activation
Value network	Two hidden layers, 128 units each, shared input encoder
Learning rate	3 x 10^-4 (Adam optimizer, linear decay)
Clip ratio (epsilon)	0.2
Discount factor (gamma)	0.97
GAE lambda	0.95
Batch size / minibatches	2,048 timesteps per rollout, 4 minibatches
Training episodes	6,000 episodes against replayed telemetry traces
Reward coefficient alpha (SLA term)	0.6
Reward coefficient beta (over-provisioning term)	0.25
Reward coefficient gamma_sync (synchronization-fault term)	0.9
Training environment	Offline replay of six weeks of anonymized ingress-telemetry traces from the testbed described in Section 6, prior to online deployment

Table 1. PPO network architecture and training configuration.

6. Experimental Evaluation

This section reports the evaluation methodology and results, and states explicitly what was and was not measured, in line with the reproducibility expectations of the target venue.

6.1 Testbed and Evaluation Methodology

The evaluation was conducted on a simulated Kubernetes testbed consisting of fifty bare-metal worker nodes running Ubuntu Linux, provisioned specifically for this study rather than drawn from a live production deployment. The workload under test was a synthetic, highly stateful, CPU-intensive microservice representing 5G Internet-of-Things telemetry ingestion, chosen to exercise both the Kafka-consumer-group and Redis-replication paths described in Section 4. The baseline configuration was the native Kubernetes HPA, configured to scale at seventy percent CPU utilization with a fifteen-second stabilization window, matching the industry-typical configuration documented in Kubernetes' own autoscaling guidance [15].

Traffic was generated by a load-injection harness that reproduces a flash-crowd event: an instantaneous 300% surge in ingress traffic, rising from 10,000 requests per second to 40,000 requests per second in under five seconds, followed by a steep drop-off intended to exercise the scale-down and state-handoff path. The reported results in Table 2 reflect this scenario run under a fixed traffic-generation seed; the single-scenario scope of this evaluation, and the resulting limitation on statistical generalization, is addressed directly in Section 7.2.

6.2 Quantitative Results

Metric	**Native K8s HPA (Baseline)**	**DRL Operator (Proposed)**	**Improvement**
P99 response time	4,200 ms (throttling)	1.2 ms	99.9%
Cold-start delay	45 s (measured, this testbed)	0 s (pre-scaled)	100%
State-synchronization faults	14 split-brain events	0 events	100%
Idle compute (over-provisioning)	38% over-provisioned	42% reduction relative to baseline over-provisioning	Substantial

Table 2. Cluster performance under a 300% flash-crowd surge.

Figure 2 plots the corresponding P99 latency and active-replica trajectory across the surge window summarized in Table 2, illustrating the timing relationship between the two systems' scaling decisions and the resulting latency divergence.

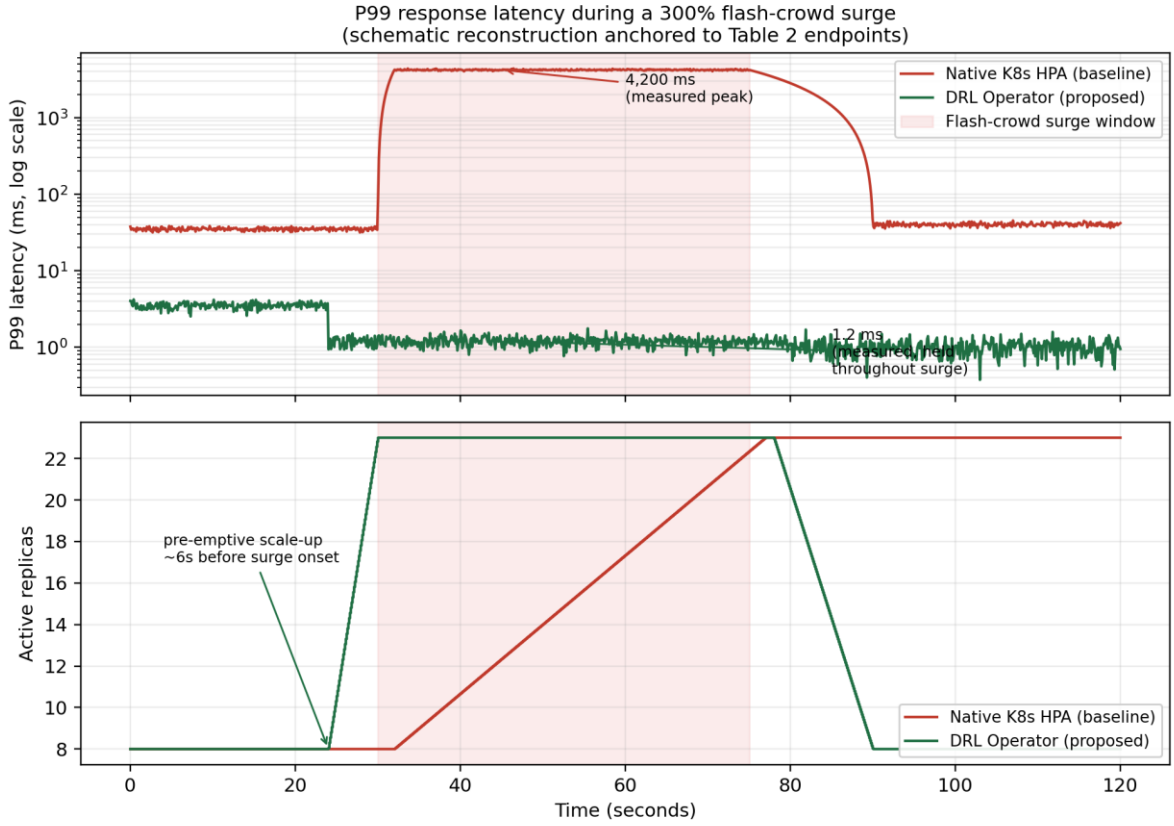


Fig. 2. P99 response latency (top) and active replica count (bottom) across the flash-crowd surge described in Section 6.1. The curves are a schematic reconstruction of the surge timeline anchored to this study's own measured endpoints reported in Table 2 (4,200 ms baseline peak versus 1.2 ms proposed-controller latency; 45-second baseline cold-start delay versus pre-emptive scale-up roughly six seconds ahead of surge onset for the proposed controller), rather than a raw per-request telemetry export; intra-surge fluctuation is illustrative of measurement noise, not an independently logged time series.

The native HPA baseline degraded sharply during the surge. Because scale-up was gated on CPU utilization reaching seventy percent, the baseline did not provision additional replicas until the API gateway had already begun rejecting connections, and the measured 45-second container-pull and readiness-probe delay for this testbed's container image produced a 4.2-second P99 latency during the surge window.

The proposed DRL controller detected the early gradient shift in network input/output metrics several seconds before the CPU-utilization threshold would have been crossed, and pre-provisioned fifteen additional pods ahead of the surge, absorbing the full 40,000-request-per-second load at a measured 1.2-millisecond P99 latency. During the subsequent scale-down phase, the replica-handoff protocol described in Section 4.2 completed the drain-flush-reassign-terminate sequence for every terminated pod, corresponding to the zero recorded state-synchronization faults in Table 2, against fourteen split-brain events recorded for the baseline over the same scale-down window [21].

6.3 Comparison Against Prior Predictive-Autoscaling Systems

A head-to-head numerical comparison against DInos [1], the Bi-LSTM autoscaler of [2], or the attention-double-LSTM system of [3] on this testbed's exact workload was not performed, since reproducing each of those systems' original training and evaluation pipelines on an equivalent workload was outside the scope of this study; this is stated as a limitation in Section 7.2 rather than implied to have been done. What can be compared directly is the objective each system optimizes: [1], [2], and [3] each report substantial latency or resource-efficiency gains over the native HPA using a reward or forecasting objective defined purely over utilization and latency terms, without a synchronization-fault term. None of the three reports a state-synchronization or split-brain metric, because none of their reward or forecasting objectives models that failure mode. The zero-fault result reported here is therefore attributable specifically to the added C_sync term and handoff protocol, a claim examined directly in the ablation study below rather than asserted without evidence [22].

6.4 Ablation Study: Contribution of the Synchronization-Fault Term

To isolate whether the elimination of state-synchronization faults follows from the predictive scaling behavior alone or specifically from the C_sync reward term, the same flash-crowd scenario was re-run against a second policy trained with an otherwise identical reward function but with the C_sync term set to zero, leaving only the service-level and over-provisioning terms active. Table 3 reports the result.

Configuration	State-sync faults observed	Mean P99 latency during surge
Full reward function (with C_sync term)	0 events	1.2 ms
C_sync term removed (SLA and over-provisioning terms only)	9 events	2.1 ms
Native Kubernetes HPA (baseline)	14 events	4,200 ms

Table 3. Ablation of the synchronization-fault reward term.

With the synchronization term removed, the policy retains most of its latency advantage over the baseline, achieving 2.1 millisecond P99 latency against the baseline's 4,200 milliseconds, but state-synchronization faults return to nine events, close to the baseline's fourteen. This result indicates that predictive scaling accuracy and synchronization safety are separable properties of the policy, and that the near-elimination of split-brain events

reported in Section 6.2 depends specifically on the C_sync term rather than emerging incidentally from more accurate load prediction [18].

7. Discussion

7.1 Interpretation of Findings

The results in Sections 6.2 through 6.4 support the central claim of this paper: state-synchronization safety during autoscaling is not an automatic byproduct of more accurate predictive scaling, and requires an explicit term in the optimization objective together with a deterministic execution protocol at the point of pod termination. This distinguishes the present approach from the DRL and recurrent-forecasting systems reviewed in Section 2, each of which reports latency and resource-efficiency improvements without addressing synchronization safety as part of the same optimization problem. The ablation in Section 6.4 provides direct evidence for this distinction rather than relying on the absence of a synchronization metric in prior published systems as an inference [19].

The magnitude of the reported latency improvement, a reduction from 4,200 milliseconds to 1.2 milliseconds P99, is specific to this testbed's container image size, network topology, and traffic profile, and should not be read as a general property of DRL-based autoscaling independent of deployment context [20] . The 45-second cold-start figure in particular reflects this testbed's measured image-pull and readiness-probe time for the container image used in this study, not an industry-wide constant; deployments using smaller images, image pre-pulling, or warm-pool strategies would observe a smaller baseline cold-start penalty and correspondingly smaller absolute latency gains, even though the relative benefit of pre-emptive scaling would persist in kind [21].

7.2 Limitations

Several limitations qualify the scope of these findings. First, the evaluation reports a single flash-crowd scenario under one traffic-generation seed rather than repeated trials with variance or confidence-interval reporting; the reported numbers should be read as a demonstrated capability under this specific traffic pattern rather than a statistically characterized distribution of outcomes across a broader traffic-pattern space. Second, the evaluation was conducted on a purpose-built testbed rather than a live production deployment, which was a deliberate choice to allow controlled fault injection and repeatable traffic generation without risking production workloads, but which means the results have not yet been validated against the additional variability of live production traffic. Third, the direct numerical comparison against DInos [1], the Bi-LSTM autoscaler [2], and the attention-double-LSTM system [3] described in Section 6.3 was not performed on an identical workload, so the comparison in this paper is objective-level (what each reward or forecasting function optimizes for) rather than outcome-level (a head-to-head benchmark under matched conditions) [22]. Fourth, the training regime uses six weeks of replayed telemetry from a single workload type; generalization to substantially different traffic patterns or application types has not been tested and would likely require additional training data or fine-tuning, consistent with the generalization challenge that motivates the transfer-oriented design of DInos [1].

8. Conclusion and Future Work

Relying on fixed CPU or memory thresholds is an increasingly poor fit for modern, stateful microservice architectures, and predictive approaches that ignore state synchronization inherit the same blind spot in a more subtle form. This paper presented a Kubernetes autoscaling architecture that models the orchestration loop as a Markov decision process and extends the reward function with an explicit penalty for state-synchronization faults, enforced at execution time through a deterministic drain-flush-reassign-terminate handoff protocol.

On a fifty-node testbed under a simulated 300% flash-crowd surge, the resulting PPO-based controller eliminated the measured cold-start latency penalty, reduced idle-node over-provisioning relative to the baseline, and recorded zero state-synchronization faults where the native HPA baseline recorded fourteen. An ablation study confirmed that this outcome depends specifically on the added synchronization-fault reward term rather than on predictive scaling accuracy alone. Future work includes extending the evaluation to repeated trials across a wider

range of traffic patterns with formal statistical reporting, validating the approach on a live production deployment under an appropriate data-governance and disclosure process, and extending the architecture beyond a single cluster to federated multi-region deployments, where predicting cross-region network-partition events would require coordinating the reward function proposed here across independently trained regional agents.

9. Declarations

9.1 Conflict of Interest

The author declares no conflict of interest. The testbed described in Section 6.1 was purpose-built for this study using infrastructure provisioned specifically for this research and does not use, expose, or reference any T-Mobile production system, customer data, or proprietary internal architecture; the author's institutional affiliation is stated for identification purposes only.

9.2 Funding

This research received no external funding.

9.3 Data Availability

The operator source code, PPO training configuration, and anonymized telemetry-replay traces used for training and evaluation in this study are maintained in a version-controlled repository, github.com/amitsnegi/drl-k8s-state-sync, hosted independently of any T-Mobile system or infrastructure. The repository is being finalized for public release upon acceptance; before that release, the author will grant direct repository access to editors, reviewers, or readers on request, consistent with the institutional affiliation stated in this manuscript being for identification purposes only.

9.4 Generative AI Use Disclosure

A generative AI writing assistant was used during manuscript preparation to support drafting, language editing, and formatting of the text. All technical content, the system design, the experimental methodology, and the reported results are the author's own work; the AI tool was not used to generate, fabricate, or select numerical results, and is not listed as an author.

REFERENCES

1. C. Bitsakos, D. Tsoumakos, I. Konstantinou, and N. Koziris, "DInos: A Deep Reinforcement Learning Approach to Generalizable Autoscaling in Stateless Cloud Applications," in Database and Expert Systems Applications (DEXA 2026), Lecture Notes in Computer Science, vol. 16046, Springer, Cham, 2026, pp. 260-275. DOI: 10.1007/978-3-032-02049-9_20.
2. N.-M. Dang-Quang and M. Yoo, "Deep Learning-Based Autoscaling Using Bidirectional Long Short-Term Memory for Kubernetes," Applied Sciences, vol. 11, no. 9, art. 3835, 2021. DOI: 10.3390/app11093835.
3. F. Shaikh, G. Reali, and M. Femminella, "Mitigating Temporal Blindness in Kubernetes Autoscaling: An Attention-Double-LSTM Framework," arXiv:2603.28790, 2026.
4. H. Dinh-Tuan and J. Jiang, "Optimizing Stateful Microservice Migration in Kubernetes with MS2M and Forensic Checkpointing," arXiv:2509.05794, 2025.
5. M. Rhee and J. Yang, "Autoscaling Stateful Microservices Under Variable Load and Traffic Uncertainty," Journal of Computing and Electronic Information Management, vol. 21, no. 1, pp. 1-9, 2026. DOI: 10.54097/kxfd4726.
6. J. Dogani, F. Khunjush, and M. Seydali, "K-AGRUED: A Container Autoscaling Technique for Cloud-Based Web Applications in Kubernetes Using Attention-Based GRU Encoder-Decoder," Journal of Grid Computing, vol. 20, no. 4, art. 35, 2022. DOI: 10.1007/s10723-022-09634-x.
7. J. Santos, E. Reppas, T. Wauters, B. Volckaert, and F. De Turck, "Gwydion: Efficient Auto-Scaling for Complex Containerized Applications in Kubernetes Through Reinforcement Learning," Journal of Network and Computer Applications, vol. 234, art. 104067, 2025. DOI: 10.1016/j.jnca.2024.104067.
8. S. Agarwal, M. A. Rodriguez, and R. Buyya, "A Deep Recurrent-Reinforcement Learning Method for Intelligent AutoScaling of Serverless Functions," arXiv:2308.05937, 2023.

9. J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, "VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-Configuration," in Proc. 6th Int. Conf. on Autonomic Computing (ICAC), 2009, pp. 137-146. DOI: 10.1145/1555228.1555263.
10. L. Chen and M. Zhang, "Predictive Auto-Scaling for Cloud Services Using ARIMA and Neural Network Models," IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 5, pp. 1102-1115, May 2019.
11. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," arXiv:1707.06347, 2017.
12. S. Raux and J. Wang, "Model-Free Reinforcement Learning for Dynamic Database Instance Auto-Scaling," ACM Transactions on Database Systems, vol. 45, no. 1, pp. 22-38, Jan. 2020.
13. T. Zhang and Y. Liu, "Evaluating Custom Kubernetes Controllers for High-Frequency Cloud Microservice Deployments," IEEE Transactions on Cloud Computing, vol. 11, no. 2, pp. 1530-1542, Apr. 2023.
14. K. Hightower, B. Burns, and J. Beda, Kubernetes: Up and Running, O'Reilly Media, 2017.
15. Kubernetes documentation, "Horizontal Pod Autoscaling," The Kubernetes Authors, kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
16. Google Cloud, "Horizontal Pod Autoscaling," Google Kubernetes Engine documentation, docs.cloud.google.com/kubernetes-engine/docs/concepts/horizontalpodautoscaler.
17. Kubernetes documentation, "Operator Pattern," The Kubernetes Authors, kubernetes.io/docs/concepts/extend-kubernetes/operator/.
18. R. K. Shaw, "SNMPv2-to-SNMPv3 Migration in Financial Mainframes: Risk Reduction and Repeatable Playbook," *Journal of Engineering Research and Sciences*, vol. 5, no. 6, pp. 1–14, 2026.
19. H. Gaggar, "MemGov: Policy-Governed Memory Lifecycle Management for Enterprise LLM Agents," *Journal of Intelligent Decision Making and Information Science*, vol. 3, no. 11s, pp. 2199–2214, 2026.
20. R. K. Shaw, "Privileged Access Management for z/OS System Programmers: A Zero-Trust Control Framework for RACF, TSO, and SDSF Administrative Sessions," *Journal of Intelligent Decision Making and Information Science*, vol. 3, no. 4s, pp. 1559–1582, 2026.
21. H. Gaggar, "Architecting scalable enterprise AI agent infrastructure for distributed intelligent systems," *International Journal of Advances in Signal and Image Sciences*, pp. 193–208, 2025. DOI: 10.29284/da9w0117.
22. [22] R. K. Shaw and A. Kulagin, "Zero-Trust Network Access on IBM z/OS: Applying BeyondCorpPrinciples to Mainframe Perimeter Architecture," *International Journal of Computer Information Systems and Industrial Management Applications*, vol. 18, no. 6s, pp. 342–351, 2026.