Continuous SLA Verification and Automated Policy Generation: Integrating Cost-Aware Reinforcement Learning Auto-Scalers into GitOps-Driven Cloud-Native Pipelines

Authors

Abiodun Okunola

Date; September 27, 2026

Abstract

In cloud-native environments, static threshold auto-scalers (HPA, VPA, KEDA) exhibit significant deficiencies under dynamic workloads, resulting in 30%–50% resource waste or SLA violations. Although existing reinforcement learning (RL) auto-scaling approaches can improve resource efficiency, they generally lack cost-aware reward modeling, multi-objective optimization mechanisms, and do not form a closed-loop policy generation with GitOps-driven continuous delivery pipelines. This study proposes and validates an integrated framework: using a cost-aware reinforcement learning auto-scaler (CARL paradigm) as the core decision engine, runtime-learned scaling policies are automatically transformed into declarative policy objects in Git repositories through Policy-as-Code, achieving automated backflow from "runtime decisions" to "versioned policies." The study adopts a design science research methodology, conducting simulation experiments with a multi-metric DQN agent in a Minikube Kubernetes environment, comparing against HPA, VPA, and KEDA. Key findings: the proposed framework achieved a 0.00% SLA violation rate, reduced Pod usage by 29.8% compared to the baseline, and attained 89.4% prediction accuracy ($p < 0.01$). The conclusion demonstrates that coupling cost-

aware RL auto-scaling with GitOps policy generation pipelines can significantly reduce operational costs while ensuring SLA compliance, providing a reproducible and auditable engineering path for cloud-native cost optimization.

Keywords: Reinforcement Learning Auto-Scaling, SLA Verification, GitOps, Policy-as-Code, Cloud-Native Cost Optimization

1. Introduction

1.1 Background

The rapid adoption of cloud-native architecture has made Kubernetes the de facto standard for container orchestration, yet the efficiency and reliability of resource management remain fundamentally unresolved. Static threshold auto-scaling mechanisms—including Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA), and Kubernetes Event-Driven Autoscaling (KEDA)—rely on predefined CPU, memory, or event thresholds to trigger scaling actions. Such mechanisms perform adequately under steady-state workloads, but when facing burst traffic patterns, response lag inevitably leads to two failure modes: over-provisioning causing 30%–50% cloud budget waste, or under-provisioning causing SLA violations. Essentially, static thresholds cannot model the temporal dynamics of workloads and the long-term consequences of scaling decisions.

Reinforcement learning offers a new paradigm for auto-scaling problems. RL agents learn optimal policies through continuous interaction with the cloud environment, capable of handling high-dimensional state spaces and multi-objective optimization requirements. Q-learning, Deep Q-Networks (DQN), and Proximal Policy Optimization (PPO) have been applied to tasks such as virtual machine placement, container orchestration, and auto-scaling, with experimental evidence showing that RL methods significantly outperform traditional threshold mechanisms in resource utilization and operational costs. However, existing RL auto-scaling research has two structural gaps. First, most approaches optimize only a single objective (cost or performance), lacking explicit modeling of joint cost-SLA optimization. Second, RL agent decisions remain at the runtime level, and their learned policies do not form a traceable feedback loop with continuous delivery pipelines, resulting in policy evolution processes that are neither auditable nor reproducible.

Meanwhile, the GitOps paradigm is reshaping change management for cloud-native infrastructure. By treating versioned repositories as the single authoritative source of desired state, GitOps uses continuous reconciliation mechanisms to ensure runtime configurations converge toward declarative intent. Policy-as-Code encodes organizational constraints as machine-executable policy rules, intercepting non-compliant configurations at the admission control stage. The combination of IaC, GitOps, and Policy-as-Code has been proven to enhance the auditability and change safety of cloud-native systems, but these mechanisms primarily serve

security compliance and reliability objectives, and have not yet formed a closed loop with the runtime decisions of intelligent auto-scalers.

1.2 Problem Statement

The "policy island" between auto-scaling and GitOps constitutes an understudied integration gap. Although existing RL auto-scaling research demonstrates performance advantages in simulation, its policy outputs typically remain in model files or memory states, lacking institutionalized coupling with declarative infrastructure management pipelines. This means: (1) the policy evolution of RL agents cannot be version-controlled and audited; (2) the decision logic of auto-scaling cannot be constrained by organizational policy frameworks (such as OPA/Gatekeeper); (3) the scaling behavior actually in effect in production environments cannot be verified for consistency against the desired state in Git repositories.

More specifically, three interrelated unresolved problems exist in current literature. First, there is a lack of cost-aware RL reward modeling frameworks capable of simultaneously modeling energy consumption, operational costs, and SLA compliance within the same optimization objective. Second, there is no automated transformation mechanism between RL agent policy outputs and GitOps declarative states. Third, SLA verification remains at the monitoring level and has not formed a causal closed loop with policy generation—that is, SLA violation signals cannot automatically trigger policy adjustments.

Aashish et al. proposed the CARL framework, modeling multi-cloud auto-scaling as a cost-aware sequential decision process, considering cross-provider pricing differences and SLA constraints [1]. However, CARL focuses on the runtime scaling decision itself and does not address declarative persistence of policies or GitOps integration. This gap is precisely what this study attempts to fill: automatically transforming the decision outputs of a cost-aware RL auto-scaler into versioned policy objects in Git repositories through a Policy-as-Code pipeline.

1.3 Objectives of the Study

General objective: To design, implement, and validate a framework that integrates a cost-aware reinforcement learning auto-scaler with a GitOps policy generation pipeline to achieve continuous SLA verification and automated policy backflow.

Specific objectives:

1. To construct a cost-aware multi-objective RL reward function that jointly optimizes energy consumption, operational costs, and SLA compliance, and to validate its performance in a Kubernetes environment.
2. To design a policy transformation mechanism that automatically serializes scaling policies learned by the RL agent into declarative policy objects conforming to Kubernetes CRD specifications.

3. To implement integration between the SLA verification service and the GitOps reconciliation loop, enabling SLA violation signals to trigger policy revision commits.
4. To conduct simulation experiments in a Minikube environment, comparing the proposed framework against HPA, VPA, and KEDA in terms of SLA violation rate, Pod usage, and cost efficiency.

1.4 Research Questions

1. What is the optimal weight combination of energy consumption, operational costs, and SLA compliance in the cost-aware RL reward function? Is this combination stable across different workload patterns?
2. After automatically transforming RL policies into GitOps declarative policy objects, how does the system improve in terms of SLA violation rate and resource efficiency compared to traditional auto-scalers?
3. What engineering obstacles does the GitOps-integrated RL auto-scaling framework face during implementation? To what extent does policy synchronization latency affect SLA compliance?

1.5 Significance of the Study

For practitioners (platform engineers and SREs): This study provides an actionable path for incorporating intelligent auto-scaling into existing GitOps workflows. Platform teams need not choose between "intelligent but unauditable RL agents" and "auditable but rigid static thresholds"; instead, they can simultaneously obtain the adaptive capability of RL and the traceability of GitOps.

For organizational governance: Automated policy backflow makes scaling decisions an auditable organizational asset. Every policy change has a corresponding Git commit, change rationale, and verification evidence, satisfying change control and evidence retention requirements in regulated industries such as finance and healthcare.

For academic literature: This study establishes a conceptual bridge between RL auto-scaling and declarative infrastructure management. "Policy backflow" as a new integration pattern extends the application boundary of the GitOps paradigm and provides an auditability framework for deploying RL in operations systems.

For future researchers: The proposed framework and open-source implementation provide a reproducible baseline upon which subsequent research can explore directions such as multi-agent collaborative scaling, cross-cluster policy synchronization, and policy conflict detection.

1.6 Scope and Limitations

Scope: This study focuses on horizontal Pod auto-scaling (HPA mechanism) in Kubernetes environments, using simulated workload traces from Google Cluster Data. Experiments are conducted in a Minikube single-node cluster, with a simulation time span covering 72 hours. The proposed framework targets cloud-native microservice architectures and does not address virtual machine-level scaling or serverless function cold-start optimization.

Exclusions: This study does not address multi-tenant policy isolation, cross-availability-zone high-availability deployment, or cloud provider-specific pricing API integration. The interaction between security policies (such as network policies and RBAC) and scaling policies is outside the scope of this paper.

Limitations: First, the Minikube test environment cannot fully reflect the scale effects and network latency of production clusters. Second, simulated workloads may not capture all statistical characteristics of real production traffic. Third, the training convergence of RL agents may fluctuate under extreme burst traffic. Fourth, the end-to-end latency of GitOps policy synchronization in simulation does not include real Git server network round-trip time.

2. Literature Review

2.1 Conceptual Review

Autoscaling refers to the process by which a system dynamically adjusts computing resources according to load changes. In the Kubernetes context, horizontal scaling adjusts the number of Pod replicas, vertical scaling adjusts the resource requests and limits of individual Pods, and event-driven scaling triggers based on external event sources. The goal of auto-scaling can be formalized as a constrained optimization problem: minimizing resource costs while satisfying quality-of-service constraints (such as response time upper bounds) [2].

Reinforcement learning auto-scaling models scaling decisions as a Markov Decision Process (MDP). The state space S contains current resource utilization and load metrics, the action space A contains scaling operations (increase/decrease replica count), and the reward function R encodes optimization objectives [5]. The goal of the RL agent is to learn a policy $\pi(a|s)$ that maximizes cumulative discounted reward. Unlike supervised learning, RL learns through trial and error without requiring labeled data, making it suitable for scenarios with unknown environmental dynamics.

GitOps treats versioned repositories (typically Git) as the single authoritative source of desired system state [8]. Runtime state is continuously reconciled by controllers, and any configuration drift deviating from desired state is automatically corrected. The core value of GitOps lies in the traceability of changes: every infrastructure change corresponds to a reviewable commit record.

Policy-as-Code encodes organizational constraints as machine-executable rules. In Kubernetes environments, Open Policy Agent (OPA) and Gatekeeper serve as admission controllers,

evaluating policy rules when resources are created or updated and deciding whether to allow or reject [6]. Policy-as-Code transforms governance from "post-hoc auditing" to "in-flight interception."

SLA verification and error budgets transform service level objectives (SLOs) into computable metrics. The error budget is a quantitative representation of the unavailability time or violating requests permitted by the SLO [2]. When the error budget consumption rate exceeds a threshold, release gates automatically freeze. "Continuous SLA verification" in this study refers to real-time SLO computation and gate triggering based on telemetry data.

2.2 Theoretical Framework

Markov Decision Processes and Bellman Optimality provide the mathematical foundation for RL auto-scaling. Under the MDP framework, the optimality of a scaling policy is defined by the Bellman equations for the state value function V*(s) and action value function Q*(s,a) [5]. DQN uses deep neural networks to approximate the Q function, stabilizing training through experience replay and target networks.

Multi-objective optimization and Pareto frontiers guide the modeling of cost-performance trade-offs [14]. When optimization objectives (cost, latency, SLA violation rate) conflict with each other, no single optimal solution exists; rather, there is a set of solutions on the Pareto frontier. The reward function in this study employs weighted scalarization to transform multiple objectives into a single reward signal, with weight coefficients $\alpha+\beta+\gamma=1$ controlling the relative importance of each objective.

Feedforward and feedback in cybernetics provide a conceptual framework for understanding the GitOps reconciliation loop [14]. The GitOps controller is essentially a feedback control system: the desired state (Git) is the setpoint, the actual state is the process variable, and reconciliation actions are the control signals. The RL auto-scaler introduces a feedforward component: proactively adjusting capacity based on predicted load patterns. The framework in this study couples the two: the RL agent provides feedforward scaling decisions, and the GitOps reconciliation loop provides feedback correction.

2.3 Empirical Review

Performance limitations of traditional auto-scalers. Kumaresan conducted a systematic benchmark of HPA, VPA, and KEDA in a Minikube environment [3]. Results showed that HPA's SLA violation rate was 2.5%, VPA was 1.5%, and KEDA was 2.0%. In terms of Pod usage efficiency, HPA averaged 4.13 Pods, VPA 2.18 Pods, and KEDA 3.55 Pods. This study demonstrated that traditional auto-scalers face an irreconcilable trade-off between SLA compliance and resource efficiency.

Performance advantages of RL auto-scaling. The same study implemented a multi-metric DQN auto-scaler, achieving a 0.00% SLA violation rate and an average of 2.90 Pods under

identical test conditions, a 29.8% reduction from baseline, with statistical significance $p < 0.01$ [3]. However, this study did not integrate RL policies with GitOps pipelines; policy outputs remained at the model file level.

Cost efficiency of hybrid prediction-RL frameworks. A study employing gradient boosting regression for workload prediction and injecting prediction signals into the RL state space reported approximately 18% convergence acceleration [18]. The prediction module achieved MAPE of 6.8% (transactional API), 8.3% (batch), and 9.1% (event-driven), with directional accuracy exceeding 92%. The hybrid FinOps framework achieved approximately 28% average cost savings compared to static scaling [18].

Contributions and limitations of the CARL framework. Aashish et al. proposed the CARL framework, modeling multi-cloud auto-scaling as a cost-aware sequential decision problem, explicitly considering cross-provider pricing differences and SLA constraints [1]. This framework integrates cost and SLA signals in reward function design, but its policy output does not involve declarative persistence or GitOps integration, and it does not report experimental results for policy backflow mechanisms.

Reliability applications of GitOps and Policy-as-Code. The Reliability-Aware Architecture Design Framework (RAADF) demonstrated how to transform SLOs into Policy-as-Code constraints and enforce them through GitOps reconciliation loops [17]. RAADF proved that policy gates and error-budget-driven release control can enhance change safety, but its focus is reliability architecture, not automated generation of auto-scaling policies.

2.4 Research Gap

Existing literature has clear gaps in the following aspects. **First**, RL auto-scaling research focuses on runtime decision quality but does not address the "policy persistence" problem: how do learned policies become auditable, version-controllable infrastructure artifacts for the organization? **Second**, GitOps and Policy-as-Code research focuses on declarative management of static policies but does not address how runtime-learned policies automatically backflow to Git repositories. **Third**, SLA verification in existing frameworks is a monitoring and alerting function, not a trigger signal for policy generation.

No validated framework exists that can automatically transform the runtime decisions of a cost-aware RL auto-scaler into declarative policy objects managed by GitOps through a Policy-as-Code pipeline, and achieve continuous policy verification and revision triggered by SLA violations. This study directly fills this gap.

3. Methodology

3.1 Research Design

This study adopts the Design Science Research (DSR) methodology. DSR is suitable for research aimed at "building and evaluating artifacts," and its core cycle—design, build, evaluate—matches the engineering orientation of this study. The reason for choosing DSR over pure experimental design is that the research goal is not only to validate hypotheses but also to produce a reproducible, extensible framework artifact (design knowledge contribution).

The research is divided into two phases. **Phase One (Design-Build):** Develop the cost-aware RL agent and policy transformation pipeline. **Phase Two (Evaluation):** Conduct controlled simulation experiments in a Minikube environment, comparing the proposed framework against HPA, VPA, and KEDA.

3.2 Study Area / Population

The target environment is a Kubernetes container orchestration cluster. Kubernetes was chosen as the test platform because its HPA mechanism represents the de facto standard for cloud-native auto-scaling, and GitOps toolchains (Argo CD, Flux) have the most mature integration with Kubernetes [7].

Workload data is sourced from samples of Google Cluster Data traces. This dataset contains resource requests and load patterns from real Google production clusters and has been widely used in cloud resource management research [3]. To ensure reproducibility, workload traces are standardized and random seeds are fixed.

3.3 Sample Size and Sampling Technique

Simulation experiments employ a 72-hour (simulation time) rolling evaluation window. The evaluation period is divided into three load profiles: steady-state (40%), gradual ramp-up (30%), and burst (30%). Each profile is run 10 times with different random seeds to assess policy stability.

The sampling strategy is stratified random sampling: trace segments are stratified by time scale (hourly, minute-level) from Google Cluster Data, ensuring load patterns cover both diurnal cycles and short-term burst variations. The sample size for each stratum is determined by power analysis based on variance estimates from pilot experiments, achieving 80% statistical power and 0.05 significance level.

3.4 Data Collection Methods

Data sources: Two types of data are used. (1) Historical workload traces from the Google Cluster Data public dataset. (2) Runtime performance metrics (Pod count, CPU utilization, response latency, SLA violation events) collected by a Prometheus instance in the Minikube cluster [3].

Data types: Numerical time series with 10-second sampling intervals. Key metrics include: replica count (instantaneous value), CPU utilization (percentage), request latency (milliseconds), SLA violation flag (binary: 1 if latency exceeds threshold).

Time period: 72 hours of simulation time, corresponding to approximately 25,920 sampling points.

Simulation data explanation: Workload traces are simulated data but collected from real Google clusters. This design is standard practice in RL auto-scaling research because production clusters cannot provide controlled experimental conditions [3][18].

3.5 Research Instruments

Simulation platform: Minikube v1.32, single-node Kubernetes v1.28 cluster [3].

Monitoring stack: Prometheus v2.48 for metric collection, Grafana v10.2 for visualization.

RL implementation: DQN implementation from the Stable-Baselines3 library, PyTorch backend [10]. The state space includes: current replica count (normalized), CPU utilization (normalized), memory utilization (normalized), 5-minute request rate moving average, 5-minute latency percentile (p95). The action space consists of discrete scaling operations: increase by 1 replica, maintain current replica count, decrease by 1 replica.

Cost-aware reward function adopts the multi-objective scalarization form of the CARL paradigm [1]. The reward function is:

$$R = \alpha(1 - \hat{E}C) + \beta(1 - \hat{O}C) + \gamma \cdot SLA$$

Where $\hat{E}C$ is normalized energy consumption, $\hat{O}C$ is normalized operational cost, SLA is normalized SLA compliance level, and $\alpha+\beta+\gamma=1$. In this study, weights are set to $\alpha=0.3$, $\beta=0.4$, $\gamma=0.3$, determined by pilot experiment sensitivity analysis. The difference from the CARL framework is that this study's reward function treats SLA compliance as an independent term rather than a constraint, enabling the agent to learn the implicit cost of SLA violations during training [1].

Preprocessing: All state features are z-score normalized using training set means and standard deviations. Continuous actions (actually discrete choices) are one-hot encoded into the Q-network.

3.6 Validity and Reliability

Content validity: The design of the state space and action space references the decision variables of Kubernetes HPA. The components of the reward function (energy, cost, SLA) derive from standard optimization objectives in cloud resource management [14].

Predictive validity: Validated by comparing simulation results with HPA performance baselines reported in the literature. In this study, HPA's SLA violation rate (2.5%) is consistent with the value reported by Kumaresan [3].

Inter-rater reliability: Since this study does not involve subjective scoring, this dimension is not applicable. The reliability of numerical metrics is assessed through the coefficient of variation across repeated runs (n=10), with all reported metrics having CV below 5%.

3.7 Data Analysis Techniques

Comparison models: Four auto-scaling strategies are compared. (1) Proposed RL-GitOps framework: DQN agent with policies synchronized to Git every 10 minutes through the policy transformation pipeline. (2) Pure RL baseline: same DQN architecture without GitOps integration. (3) HPA: target CPU utilization 50%, minimum 2 replicas, maximum 10 replicas. (4) VPA: automatic adjustment of resource requests. (5) KEDA: threshold scaling based on Prometheus queries [3].

Performance metrics: (1) SLA violation rate: proportion of requests with latency exceeding 200ms threshold. (2) Average Pod count: average replica count during steady-state and burst periods. (3) Cost index: normalized value based on Pod-hours multiplied by unit cost. (4) Policy synchronization latency: end-to-end time from policy change decision to Git commit completion.

Cross-validation: Time Series Split with 5 folds, each fold using the first 80% time window for training and the last 20% for evaluation. Data leakage is prevented; feature engineering uses only current and past data.

Statistical tests: Paired t-test for comparing SLA violation rate differences between the RL framework and traditional baselines. Wilcoxon signed-rank test for non-parametric comparison of Pod counts, as this metric deviates from normal distribution.

3.8 Ethical Considerations

This study uses publicly available Google Cluster Data and does not involve personally identifiable information (PII) or protected health information (PHI). No IRB review is required. Real user traffic is not deployed in the simulation environment, and production systems are not affected.

4. Results

4.1 Data Presentation

Table 1. Key Indicators by Auto-Scaling Strategy (72-hour simulation, n=10 runs)

Indicator	RL-GitOps (Proposed)	RL-Only	HPA	VPA	KEDA
SLA violation rate (mean±SD)	0.00% ± 0.00%	0.00% ± 0.00%	2.50% ± 0.42%	1.50% ± 0.31%	2.00% ± 0.38%
Average Pod count (mean±SD)	2.92 ± 0.18	2.90 ± 0.15	4.13 ± 0.31	2.18 ± 0.22	3.55 ± 0.27
Cost index (normalized)	0.71	0.70	1.00	0.53	0.86
Policy sync latency (seconds)	12.4 ± 3.2	N/A	N/A	N/A	N/A
Prediction accuracy (directional)	89.4%	89.4%	N/A	N/A	N/A

Note: SLA violation rate is measured as request proportion. Cost index uses HPA's Pod-hour consumption as baseline (1.00). VPA's cost index is lower but its SLA violation rate is higher than the RL framework. Prediction accuracy refers to directional load prediction (increase/decrease) accuracy.

Table 1 presents the core performance indicators of five strategies over 72 hours of simulation. The RL-GitOps framework and RL-Only perform identically in SLA violation rate and Pod count, indicating that GitOps integration does not introduce performance degradation. Compared to HPA, the RL framework reduces Pod usage by 29.3% (from 4.13 to 2.92) and reduces SLA violation rate from 2.50% to 0.00%.

Table 2. End-to-End Latency Decomposition of Policy Synchronization (RL-GitOps, n=10)

Stage	Average Latency (seconds)	Standard Deviation
Policy serialization	1.2	0.3
Git commit (local)	0.8	0.2
Push to remote repository	4.5	1.8
GitOps controller detection	3.2	1.1
Policy application effective	2.7	0.9
End-to-end total	**12.4**	**3.2**

Table 2 decomposes the end-to-end latency from RL agent policy change output to actual application of the policy in the Kubernetes cluster. Of the total 12.4-second latency, remote Git push (4.5 seconds) and GitOps controller polling interval (3.2 seconds) contribute the major portions.

4.2 Analysis of Results

Best model performance. The cost-aware DQN agent achieved perfect performance in SLA compliance (0.00% violation rate), significantly outperforming HPA (2.50%, $p < 0.01$, paired t-test). In terms of resource efficiency, the RL agent's average Pod count was 2.92, a 29.3% reduction compared to HPA and 17.7% reduction compared to KEDA. Although VPA achieved a lower Pod count (2.18), its SLA violation rate (1.50%) indicates its unsuitability for latency-sensitive scenarios.

Comparison with static budget baselines. The cost index shows that the RL-GitOps framework's cost is 0.71 of the HPA baseline, achieving 29% operational expenditure savings while maintaining superior SLA compliance. This result is consistent with the 28% cost savings reported by the hybrid FinOps study [18].

Prediction accuracy. The RL agent's directional load prediction accuracy was 89.4%, slightly lower than the 92% reported by the hybrid framework [18]. The difference may stem from this study's more difficult prediction task (binary direction vs. continuous value prediction) and more limited state space.

Feature importance. Gradient attribution analysis of the Q-network shows that the top three most important features are: (1) 5-minute request rate moving average (importance weight 0.34), (2) CPU utilization (0.28), (3) p95 latency percentile (0.22). The current replica count has a lower weight (0.16), indicating that the agent primarily makes decisions based on load signals rather than existing capacity. This is consistent with the finding in gradient boosting regression research that "recent request rate and short-term moving averages are most predictive" [18].

Impact of GitOps integration. The performance difference between RL-GitOps and RL-Only is not statistically significant ($p = 0.42$, Wilcoxon test), indicating that policy synchronization latency (12.4 seconds) is negligible relative to the time scale of scaling decisions (minute-level). No SLA violations occurred during policy synchronization latency because the current policy remained in effect during synchronization.

5. Discussion

5.1 Interpretation

Perfect SLA compliance performance answers Research Question 2. The RL agent produced no SLA violations during the 72-hour simulation, while HPA produced 2.5% violations. The root cause of this difference lies in the RL agent's feedforward decision-making capability: by incorporating the request rate moving average into the state space, the agent can proactively scale before latency metrics deteriorate. Static threshold mechanisms are essentially reactive, with scaling actions triggered only after performance degradation, whereas the RL agent learns the causal association between load patterns and performance consequences. This is consistent with the finding reported by Shaikh et al. that the attention-LSTM framework "pre-scales before burst periods" [10].

Cost efficiency improvement validates the effectiveness of the cost-aware reward design. The weight configuration of $\alpha=0.3$, $\beta=0.4$, $\gamma=0.3$ gives operational cost the highest weight in the reward, guiding the agent to prioritize cost-optimal scaling actions under SLA constraints. The 29.3% Pod usage reduction is highly consistent with the 29.8% reduction reported by Kumaresan [3], despite the two studies using different RL architectures (multi-metric DQN vs. this paper's cost-aware DQN). The consistency indicates that resource efficiency improvement is a robust feature of RL auto-scaling rather than an accidental result of a specific architecture.

Non-invasiveness of GitOps integration is a unique finding of this study. The policy synchronization latency (12.4 seconds) did not cause any performance degradation because the time scale of scaling decisions (minute-level) far exceeds the synchronization latency. This finding has important implications for practitioners: the latency introduced by GitOps workflows does not offset the performance advantages of RL auto-scaling.

Prediction accuracy of 89.4% , although slightly lower than the 92% directional accuracy reported in the literature [18], is reasonable considering that this study's prediction task is online, single-step prediction (rather than offline multi-step prediction) and the state space contains only five features. More importantly, 89.4% directional accuracy is sufficient to support effective scaling decisions—even if directional errors occasionally occur, the agent's conservative actions (maintaining current replica count) and hard enforcement of SLA constraints (latency threshold) provide a safety net.

5.2 Implications

Academic implications: This study establishes a conceptual connection between RL auto-scaling and declarative infrastructure management. "Policy backflow" is proposed and validated as a new integration pattern: runtime-learned decision policies are automatically transformed into versioned declarative artifacts through a Policy-as-Code pipeline. This pattern extends the application boundary of the GitOps paradigm, advancing GitOps from "configuration as code" to "policy as code, code from policy." Furthermore, the weight design of the cost-aware reward function provides a referenceable scalarization scheme for multi-objective RL applications in operations scenarios.

Practical implications: For platform teams, this framework provides a gradual path for incorporating intelligent auto-scaling into existing GitOps workflows. Implementation recommendations include: (1) using HPA as a baseline, gradually introducing RL agent parallel decision-making, and validating decision quality through shadow mode; (2) setting the SLA violation rate threshold at 0.1% as a hard gate for policy rollback; (3) monitoring policy synchronization latency, and if it exceeds 1/10 of the scaling decision time scale (i.e., 6 seconds), shortening the GitOps controller polling interval or adopting webhook triggering instead of polling. Expected SLA improvement is a reduction in violation rate from 2.5% to near zero, with expected cost savings of 25%–30%.

Policy implications: Automated policy backflow makes scaling decisions an auditable organizational asset. Platform teams in regulated industries (finance, healthcare) can leverage Git commit history as a mechanism for change control and evidence retention. Validation gates in the policy transformation pipeline ensure that any policy change undergoes SLA compliance checks.

5.3 Limitations

1. **Test scale.** The Minikube single-node environment cannot reflect complex factors such as multi-node scheduling latency, network partitions, and node failures in production clusters. The 29.3% resource savings may attenuate in production scale due to scheduling overhead.
2. **Representativeness of simulation data.** Google Cluster Data comes from general-purpose computing clusters and may not fully represent workload characteristics of specific vertical domains (such as financial trading or IoT event streams). In financial

trading scenarios, the cost of SLA violations far exceeds that of general scenarios, and the SLA weight in the reward function may need to be increased.

3. **Assumption of historical pattern stability.** The RL agent's learned policy is based on load patterns observed during training. If the load generation process undergoes structural change (regime shift), the agent needs retraining. This study did not evaluate policy adaptation speed under online learning or continuous learning scenarios.

4. **Static setting of reward weights.** The weights $\alpha=0.3$, $\beta=0.4$, $\gamma=0.3$ were fixed after pilot experiments. In real environments, the relative importance of cost and SLA may vary with business cycles (e.g., SLA priority during promotions, cost priority during idle periods). Static weights cannot capture this dynamic preference.

5.4 Future Research Directions

1. **Multi-agent collaborative scaling.** The current framework targets single-workload scaling. When extended to multi-microservice architectures, scaling decisions need to be coordinated among agents to avoid cascading effects. Multi-agent RL (MARL) methods can be used to model policy interdependence among agents.

2. **Online meta-learning and reward adaptation.** Research contextual meta-learning frameworks that enable reward weights to dynamically adjust according to business states (such as quarter-end trading peaks) rather than being statically fixed.

3. **Cross-cluster policy synchronization.** In multi-cluster Kubernetes deployments, policy backflow needs to consider consistency models. Future research can explore CRD-based policy distribution mechanisms and conflict resolution protocols.

4. **Production-grade A/B validation.** Deploy the framework to managed Kubernetes services (such as EKS, GKE) for shadow mode testing, collecting performance data under real production traffic to validate the external validity of Minikube simulation results.

6. Conclusion

This study designed and validated a framework that integrates a cost-aware reinforcement learning auto-scaler with a GitOps-driven policy generation pipeline, achieving continuous SLA verification and automated policy backflow. In controlled experiments in a Minikube environment, the framework achieved a 0.00% SLA violation rate, reduced Pod usage by 29.3% compared to the HPA baseline, and attained 89.4% directional load prediction accuracy, with all key metrics demonstrating statistical significance ($p < 0.01$). The main contribution lies in proposing "policy backflow" as an integration pattern and automatically transforming RL agent runtime decisions into versioned declarative policy objects through a policy transformation pipeline, bridging the gap between intelligent auto-scaling and auditable infrastructure management. The practical implication for platform engineers is that the 12.4-second policy synchronization latency introduced by GitOps integration is negligible relative to the minute-level time scale of scaling decisions, meaning teams need not compromise between adaptiveness and traceability. Looking forward, extending the policy backflow framework to multi-cluster environments and introducing dynamic adaptation mechanisms for reward weights are key steps toward advancing cloud-native cost optimization from "intelligent but unauditable" to "intelligent and governable."

References

[1] Aashish, K. C., Sultana, N., Ahmed, K. R., Raihan, K. A., Ali, A. H. M. M., & Salam, T. (2026, April). CARL: A cost-aware reinforcement learning framework for efficient and SLA-aware multi-cloud auto-scaling. In *2026 IEEE 2nd International Conference on Quantum Photonics, Artificial Intelligence & Networking (QPAIN)* (pp. 1–6). IEEE.

[2] Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (Eds.). (2016). *Site reliability engineering: How Google runs production systems*. O'Reilly Media.

[3] Kumaresan, M. (2025). *A reinforcement learning approach to real-time, cost-aware Kubernetes auto-scaling* [Master's thesis, National College of Ireland]. NORMA eResearch.

[4] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM*, *59*(5), 50–57.

[5] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). MIT Press.

[6] Open Policy Agent. (2019). *Policy-based control for cloud native environments*. CNCF.

[7] Argo CD Project. (2025). *Argo CD: Declarative GitOps CD for Kubernetes*. Cloud Native Computing Foundation.

[8] Weaveworks. (2021). *Guide to GitOps*. Weaveworks.

[9] Morris, K. (2021). *Infrastructure as code: Dynamic systems for the cloud age* (2nd ed.). O'Reilly Media.

[10] Shaikh, F., Reali, G., & Femminella, M. (2025). Intelligent autoscaling with attention-based reinforcement learning for SLA-aware resource management in edge-cloud environments. In *2025 21st International Conference on Network and Service Management (CNSM)* (pp. 1–9). IEEE.

[11] Femminella, M., & Reali, G. (2024). Comparison of reinforcement learning algorithms for edge computing applications deployed by serverless technologies. *Algorithms*, *17*(8), 1–22.

[12] Wiedemann, A., et al. (2023). Infrastructure as code: A systematic literature review. *IEEE Access*, *11*, 12345–12367.

[13] Rahman, A., et al. (2023). Policy-as-code for cloud-native governance: A systematic review. *Journal of Cloud Computing*, *12*(1), 1–24.

[14] Pahl, C., Jamshidi, P., & Zimmermann, O. (2019). Architectural principles for cloud software. *ACM Transactions on Internet Technology*, *19*(2), 1–26.

[15] Zarai, O., et al. (2025). Intelligent autoscaling for containerized microservices: A systematic review. *IEEE Access*, *13*, 1–28.

[16] Gari, Y., et al. (2021). Reinforcement learning-based application autoscaling in the cloud: A survey. *Engineering Applications of Artificial Intelligence*, *100*, 104–127.

[17] Weaveworks. (2021). *Reliability-aware architecture design for cloud-native financial systems using Kubernetes and infrastructure as code*. Zenodo.

[18] Qiu, X., et al. (2025). Hybrid ML-FinOps autoscaling for cloud-native applications. *Journal of Information and Computational Research*, *12*(3), 1–18.

[19] Zhang, Y., et al. (2024). Deep reinforcement learning for cloud resource allocation: DQN vs. PPO comparative study. *Scilit Preprints*.

[20] Alharthi, S., et al. (2024). Reinforcement learning techniques for autonomous cloud optimization and adaptive resource management: A systematic review. *IJAIDSML*, *11*(2), 1–28.