NimbusGuard: A Novel Framework for Proactive Kubernetes Autoscaling Using Deep Q-Networks

Chamath Wanigasooriya
Department of Computer Science
Informatics Institute of Technology
Sri Lanka
chamath.2023@iit.ac.lk

Indrajith Ekanayake
Department of Computer Science
Informatics Institute of Technology
Sri Lanka
indrajith.e@iit.ac.lk

***Abstract*—Cloud native architecture is about building and running scalable microservice applications to take full advantage of the cloud environments. Managed Kubernetes is the powerhouse orchestrating cloud native applications with elastic scaling. However, traditional Kubernetes autoscalers are reactive, meaning the scaling controllers adjust resources only after they detect demand within the cluster and do not incorporate any predictive measures. This can lead to either over-provisioning and increased costs or under-provisioning and performance degradation. We propose NimbusGuard, an open-source, Kubernetes-based autoscaling system that leverages a deep reinforcement learning agent to provide proactive autoscaling. The agent's perception is augmented by a Long Short-Term Memory model that forecasts future workload patterns. The evaluations were conducted by comparing NimbusGuard against the built-in scaling controllers, such as Horizontal Pod Autoscaler, and the event-driven autoscaler KEDA. The experimental results demonstrate how NimbusGuard's proactive framework translates into superior performance and cost efficiency compared to existing reactive methods.**

***Index Terms*—Elasticity, Autoscaling, Microservices, Reinforcement Learning, Proactive Scaling**

I. Introduction

Popularity in microservice and container-based approaches brought the term cloud native to the light. Kratzke and Quint [1] defined cloud native applications as distributed, elastic systems designed to take full advantage of cloud environments. These applications are composed of small, independent, and deployable units known as microservices. The elasticity is provided through dynamic resource allocation to microservices through scaling them properly on demand [2]. Kubernetes [3] has become the de facto standard for microservice (container) orchestration, which handles elasticity with many of the previously mentioned properties [4]. Kubernetes has built-in mechanisms for dynamically allocating resources to constituent containers and scaling them on demand. However, the current approaches are reactive, meaning they adjust resources only after they detect demand within the cluster. This has proven insufficient for dynamic production workloads often leading to either under-provisioning or over-provisioning of the resources [5], [6]. An under-provisioned microservice deployment cannot handle workloads efficiently, whereas an over-provisioned deployment incurs unnecessary cost. Therefore, dynamic resource allocation that is both efficient and cost-effective remains a challenging task.

To address these limitations, this paper presents NimbusGuard, a novel, multi-modal framework for proactive Kubernetes autoscaling. It introduces predictive foresight and contextual understanding necessary for efficient cloud-native resource management. The proposed solution is threefold: the Deep Q-Network (DQN) agent learns optimal scaling policies, the Long Short-Term Memory (LSTM) network forecasts future workloads to provide temporal awareness, and the Large Language Model (LLM) cognitive agent is orchestrated as a stateful reasoning workflow, validating and refining scaling decisions. Implemented as a production-ready Kubernetes operator, our framework uses a central Model Context Protocol (MCP) server to facilitate real-time, message-driven communication between the distributed AI agents.

Key Contributions:

1) A threefold DQN–LSTM–LLM architecture for intelligent, proactive autoscaling of cloud-native applications.
2) The first use of a LangGraph-orchestrated [7] LLM agent to validate infrastructure-scaling decisions by enriching them with real-time log and policy data.

This paper is structured as follows. Section II surveys the related literature. Section III presents the proposed proactive Kubernetes autoscaling framework. Section IV explains the experimental setup, data collection, and load generation procedure. Section V analyses the results and discusses the main insights. Section VI summarises the contributions and suggests directions for future research.

II. Related Work

Elasticity in cloud-native applications is achieved through dynamic resource allocation. It accommodates end-user-driven fluctuations by adjusting storage, compute, and networking resources over time. An autoscaler usually decides how many resources an application receives, increasing or decreasing capacity in real time to match user demand [2]. Kubernetes-orchestrated microservice environments have built-in autoscaling at two different levels: At the inference level cluster autoscaler (CA) manages the elasticity property in the Nodes. At the application level, Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA), and Kubernetes Event-Driven

Autoscaler (KEDA) [8] manage the elasticity property in the Containers. HPA scales workloads horizontally by adding or removing pod replicas in response to resource metrics, whereas the VPA resizes individual pods by adjusting their CPU and memory requests and limits. Unlike HPA and VPA, KEDA is not part of the core Kubernetes distribution, it's a Cloud Native Computing Foundation (CNCF) graduated open-source extension for Kubernetes native HPA, enabling event-driven scale-out and scale-to-zero [9].

All Kubernetes built-in solutions are reactive, so autoscaling decisions are made solely from the system's current state [10]. Researchers have highlighted key issues of this reactive autoscaling [11]–[13]. Toka et al. [6] identified three main drawbacks of the current process: (1) scaling is reactive and purely observation-based, (2) scaling behavior is not adapting to the current variability of the demand, and (3) defining scaling behavior is cumbersome because it requires tuning numerous parameters. The same study proposed a Machine Learning (ML) based proactive scaling engine. Mondal et al. [11] highlighted the same issue of reactive autoscaling using CPU and memory as a metric, making HPA incapable of foreseeing upcoming workload spikes leading to Quality of Service (QoS) violations, long tail latency, and wasted resources. The authors proposed a proactive Custom Pod Autoscaler that uses a Gated Recurrent Unit (GRU) based load prediction model and a stability window to scale pods ahead of demand.

To address the aforementioned issues, researchers have explored proactive autoscaling [6], [12], [14] or hybrid autoscaling [15], [16] methods based on time series algorithms. In terms of proactive autoscaling, the literature reveals three themes of implementation: (1) Short-term demand forecasting based, (2) performance prediction based, and (3) Reinforcement Learning (RL) based. Each implementation exhibits inherent drawbacks. Predictive models, such as the GRU by Mondal et al. [11] and the Bi-LSTM by Dang-Quang and Yoo [12], improve upon HPA by forecasting workloads. However, they function as open-loop systems. They predict future demand but lack a sophisticated, closed-loop mechanism to learn from the real-world impact of their scaling decisions. RL approaches from Khaleq et al. [13] and Garí et al. [17] introduce adaptive decision-making but are often constrained. For example, Khaleq et al. [13] focus on learning optimal thresholds rather than selecting direct scaling actions, which is an indirect and less agile control method. Furthermore, these RL models, including the DQN-based scheduler from Jian et al. [18], operate on a limited set of default metrics and act as black boxes, lacking an engineered feature set and explicit reasoning capabilities. A significant gap in all reviewed literature is the absence of a cognitive validation layer. Also, they lack a mechanism to interpret unstructured data like real-time logs or to apply stateful reasoning to validate a scaling decision before execution. This exposes them to risks where a purely quantitative model might scale inappropriately during complex events like a canary deployment or a database migration [19].

III. Methodology

A. Overview

NimbusGuard framework ensures component integration between the DQN adapter, LSTM forecaster, native Kubernetes API, and Prometheus monitoring stack. The architecture, as depicted in Figure 1, summarizes the overall flow of our proposed approach, and it can be divided into the following decision pipeline. Data Collection $\rightarrow$ Feature Processing $\rightarrow$ DQN Inference $\rightarrow$ LLM Validation $\rightarrow$ Scaling Execution. This cycle operates in 15-second intervals with stabilization periods.

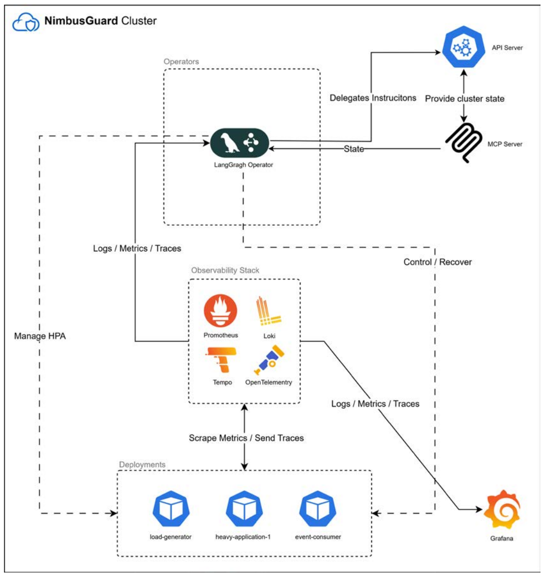

Fig. 1: High-level Overview of the Framework

B. Algorithmic Framework

NimbusGuard implements a novel hybrid autoscaling algorithm that combines Deep Q-Network (DQN), Long Short-Term Memory (LSTM) forecasting, and an optional Large Language Model (LLM) validation layer for intelligent Kubernetes container scaling. The system operates on a 30-second decision interval. At each interval, it constructs a 6-dimensional state vector. The agent's action space is a discrete set of three actions: scale_down (-1), keep_same (0), and scale_up (+1). Upon initialization, the system loads its core components:

- **DQN Networks:** A Dueling DQN architecture is used for both the main and target networks to improve policy evaluation.
- **LSTM Forecaster:** A 2-layer LSTM with 32$\rightarrow$16 hidden units is used for time-series prediction of Kubernetes consumer pod memory usage. The model uses only 2 aggregated features total_memory_mb, pod_count with a 20-interval lookback window *5 minutes* and achieves 8.7% MAPE accuracy for 15-second ahead predictions.
- **Experience Replay Buffer:** A buffer with a capacity of 10,000 experiences is used for training the DQN agent.

- **Persistent Storage:** Pre-trained models and feature scalers are loaded from a MinIO object store.

The main operational loop is orchestrated as a stateful graph using LangGraph, ensuring a modular and traceable execution of the six critical nodes at each 30-second interval.

C. Context-Aware Reward System

The context-aware reward function is designed to guide the learning agent toward optimal resource management decisions by dynamically balancing performance, efficiency, and stability. The algorithm's has the ability to adjust its reward composition based on the prevailing system context, which is determined by workload characteristics and forecast confidence.

The reward calculation begins by evaluating three components. The first is a performance score that quantifies the quality of service (QoS) using metrics. The second is an efficiency score that assesses resource utilization, rewarding configurations that meet performance targets with fewer resources. The third component is a stability penalty, introduced to discourage volatile or excessively frequent scaling actions and promote system stability.

The system's current operational state is classified by its workload level (e.g., low, nominal, or high load). This classification, along with the confidence of the workload forecast, determines two critical weights: one for the current state and one for the forecasted state. This allows the reward system to prioritize different objectives under varying conditions. For instance, during a high-load state with a high-confidence forecast, more emphasis can be placed on the proactive, forecast-based reward component.

1) **Current State Utilization Reward ($R_{\textbf{current}}$)**: This evaluates the immediate system performance using Gaussian reward curves centered on optimal resource utilization targets. It combines CPU utilization reward (target: 70%) and memory utilization reward (target: 80%) with deployment-specific normalization:

$$R_{\text{current}} = w_{\text{cpu}} \cdot R_{\text{cpu}}(u_{\text{cpu}}) + w_{\text{mem}} \cdot R_{\text{mem}}(u_{\text{mem}})$$

2) **Forecast-based Proactive Reward ($R_{\textbf{forecast}}$)**: When LSTM-based memory predictions are available, this encourages proactive scaling decisions by evaluating the forecasted system state. The forecast reward is calculated using the same utilization reward function but applied to predicted metrics:

$$R_{\text{forecast}} = R_{\text{utilization}}(\text{predicted\_metrics})$$

3) **Combined Utilization Reward**: The system weights current and forecast rewards with emphasis on predictive capabilities:

$$R_{\text{combined}} = w_{\text{current}} \cdot R_{\text{current}} + w_{\text{forecast}} \cdot R_{\text{forecast}}$$

4) **Stability and Cost Components**: Additional components include stability rewards for maintaining system health, action-specific bonuses for appropriate scaling decisions, and cost penalties for resource waste. The final reward integrates all components:

$$R_{\text{total}} = R_{\text{combined}} + R_{\text{stability}} + R_{\text{action\_bonus}} - R_{\text{cost\_penalty}}$$

The system includes action-specific bonuses for appropriate scaling (+0.2 for scale-up, +0.15 for scale-down) and penalties for unnecessary actions (-0.3 for unnecessary scaling, -0.5 for thrashing behavior).

D. LangGraph Stateful Ochestration

Algorithm 1 presents an autonomous scaling workflow implemented as a directed acyclic graph (DAG). The `ExecuteWorkflow` function initializes system state $\mathbf{S}$ with current replica count r_c and processes it through six sequential nodes:

1) `CollectMetrics`: Gathers real-time performance data
2) `GenerateForecast`: LSTM network predicts future resource demands
3) `DQNDecision`: DQN agent determines optimal scaling action
4) `ValidateDecision`: MCP server enforces replica limits, scaling velocity constraints, and mandatory cooldown periods to prevent system thrashing
5) `ExecuteScaling`: Implements the validated scaling decision
6) `CalculateReward`: Computes reward signal for DQN policy refinement

The workflow ensures system stability through MCP validation checks while enabling continuous learning via reward feedback. The function returns updated system state $\mathbf{S}$, completing the adaptive scaling cycle.

Algorithm 1 LangGraph Workflow Execution

```
Require: Current replica count $r_c$
Ensure: Updated system state $\mathbf{S}$ ExecuteWorkflow$r_c$
1: $\mathbf{S} \leftarrow$ InitializeState($r_c$)
2: $\mathbf{S} \leftarrow$ CollectMetrics($\mathbf{S}$) {Node 1: Metrics Collection}
3: $\mathbf{S} \leftarrow$ GenerateForecast($\mathbf{S}$) {Node 2: LSTM Forecasting}
4: $\mathbf{S} \leftarrow$ DQNDecision($\mathbf{S}$) {Node 3: DQN Decision Making}
5: $\mathbf{S} \leftarrow$ ValidateDecision($\mathbf{S}$) {Node 4: Safety Validation}
6: $\mathbf{S} \leftarrow$ ExecuteScaling($\mathbf{S}$) {Node 5: Scaling Execution}
7: $\mathbf{S} \leftarrow$ CalculateReward($\mathbf{S}$) {Node 6: Reward & Learning}
8: return $\mathbf{S}$ =0
```

E. Feature Engineering and Selection

To optimize the learning efficiency of the DQN agent, a feature engineering process was undertaken to reduce the state space dimensionality. The primary goal was to create a state vector that is both computationally efficient and information-rich, eliminating redundancy while retaining the most critical signals for intelligent autoscaling. This resulted in a compact 4-dimensional state vector, which distills complex system metrics into a focused representation of the deployment's current state and predicted resource needs.

The final state vector, S, is defined as:

$$S = [s_1, s_2, s_3, s_4]$$

The components are constructed dynamically using real-time metrics and predictive modeling:

s_1: **Predicted Memory Utilization (%).** The forecasted memory utilization percentage based on LSTM time-series prediction, providing proactive insight into future memory pressure. This is the primary predictive signal for scaling decisions.

s_2: **Current CPU Utilization (%).** The instantaneous CPU utilization percentage relative to the deployment's total CPU limits, calculated as:

$$s_2 = \frac{\text{Total Current CPU Usage}}{\text{Total CPU Limit}} \times 100$$

s_3: **Current Memory Utilization (%).** The instantaneous memory utilization percentage relative to the deployment's total memory limits, calculated as:

$$s_3 = \frac{\text{Total Current Memory Usage}}{\text{Total Memory Limit}} \times 100$$

s_4: **Current Replica Count.** The absolute number of currently active replicas, providing direct awareness of the current scaling state and serves as a baseline for scaling actions.

This approach for feature engineering ensures that the agent operates on the most essential signals while maintaining the ability to make informed, proactive scaling decisions. The focus on memory prediction as the primary forward-looking signal reflects the critical importance of memory management in containerized environments, where memory pressure can lead to pod evictions and service degradation.

IV. Experimental Setup

This section details the environment, application, and methodologies used to conduct a comparative analysis of the three autoscaling configurations.

A. Testbed Environment

All experiments were conducted on a MacBook Pro equipped with an Apple M4 Pro processor and 24GB of unified memory. The testbed utilized Docker Desktop for Mac (v4.x), which provided the containerization runtime. The Kubernetes environment was a KinD (Kubernetes-in-Docker) cluster running Kubernetes v1.28, provisioned and managed by Docker Desktop. A significant portion of the host machine's resources (specifically, 8 vCPU cores and 16GB of memory) were allocated to the Docker Desktop virtual machine to ensure the KinD cluster had sufficient and stable resources for the experiment. The entire test was executed within a dedicated Kubernetes namespace to ensure workload isolation.

B. Target Application

The workload consisted of a stateless, containerized application developed in Python with the FastAPI framework. The application was designed as a deterministic consumer, where each incoming request triggers a predictable and consistent amount of CPU and memory usage. This deterministic behavior makes it an ideal testbed for evaluating and comparing the responses of different autoscaling mechanisms. Each container replica was configured with the following resource specifications:

- CPU Request: 600m (0.6 of a virtual core)
- CPU Limit: 1000m (one virtual core)
- Memory Request: 512Mi
- Memory Limit: 1Gi

This configuration ensures that CPU utilization is the primary scaling signal, providing a clear metric for the autoscalers to act upon.

C. Autoscaling Configurations

Three autoscaling configurations were evaluated, representing proactive, reactive, and event-driven paradigms:

TABLE I: Comparison of the three autoscaling configurations evaluated.

Autoscaler	Paradigm	Key Configuration Details
NimbusGuard	Proactive	Uses a DQN agent with LSTM memory forecasting and a 4-dimensional state vector.
HPA	Reactive	A standard Kubernetes baseline triggered by 70% CPU or 80% memory usage, with a 30-second stabilization window.
KEDA	Event-driven	Configured to use Prometheus metrics with a 30-second polling interval and cooldown period to match HPA.

D. Load Generation and Procedure

We employed a fire-and-forget asynchronous load testing methodology [20] to simulate realistic, unconstrained traffic. This approach prevents the load generator from becoming a bottleneck and allows the system's true performance under pressure to be observed.

A custom Python script utilizing ‘asyncio‘ [21] was used to generate the load. To ensure deterministic and perfectly reproducible traffic patterns for fair comparison across all three autoscalers, the load generation process was initialized with a fixed seed.

The experiment followed a phased procedure:

- Phase 1: Ramp-up: A gradual increase in load (4 concurrent users, 40 total requests) to test the initial responsiveness of each autoscaler.
- Phase 2: Sustained Load: A period of consistent, high load (8 concurrent users, 60 total requests) to evaluate steady-state behavior and stability.

- Phase 3: Peak Load: A significant spike in traffic (15 concurrent users, 90 total requests) to challenge the system's maximum scaling capabilities and stress resilience.
- Phase 4: Cooldown: A cessation of traffic with a light load (3 concurrent users, 30 total requests) to observe scale-down behavior and resource de-allocation efficiency.

E. Data Collection and Metrics

System-wide metrics were collected using a Prometheus monitoring instance deployed within the cluster, configured with a scrape interval of 15 seconds for high-resolution data. The primary metric analyzed for this study was the number of active application replicas over the duration of the experiment. This metric reflects the scaling decisions made by each controller in response to the identical, reproducible load pattern.

V. Results

A. Performance Metrics Comparison

The results obtained by subjecting each autoscaler to an identical, phased load pattern (discussed in the load generation section) reveal a clear divergence between the different systems. The DQN-based NimbusGuard demonstrated a highly aggressive, performance-oriented strategy, while HPA and KEDA exhibited a more conservative, resource-efficient approach. This aggressive strategy is a direct consequence of the hyperparameters chosen for the DQN agent, which were tuned to prioritize future performance. By assigning a heavy weight to forecasted metrics $forecastweight = 0.7$. Specifically, a high discount factor $gamma = 0.99$ makes the agent farsighted, encouraging it to scale up proactively now to prevent future negative rewards associated with performance degradation. This forward-looking behavior is coupled with a setup designed for agility; a relatively high learning rate $lr = 0.001$ and a small replay buffer $buffer = 10000$ allow the agent to adapt quickly to the most recent workload trends. This combination results in a responsive agent that aggressively provisions resources to optimize for future Quality of Service, setting it apart from the more conservative, reactive baselines.

As shown in Table II, NimbusGuard operated with the highest average replica count (5.44), significantly more than HPA (3.05) and KEDA (2.93). This led to it having the largest resource integral (2,775 pod-seconds), indicating a strategy that prioritizes Quality of Service and responsiveness over minimizing cost. Furthermore, NimbusGuard was the most agile and least stable system, executing 8 total scaling events, double that of HPA and KEDA (4 events each). In contrast, HPA and KEDA offered greater stability and resource efficiency, making them more cost-effective but potentially less responsive to sudden load spikes. These findings highlight a fundamental trade-off: the proactive, performance-focused scaling of NimbusGuard versus the reactive, cost-efficient stability of traditional autoscalers.

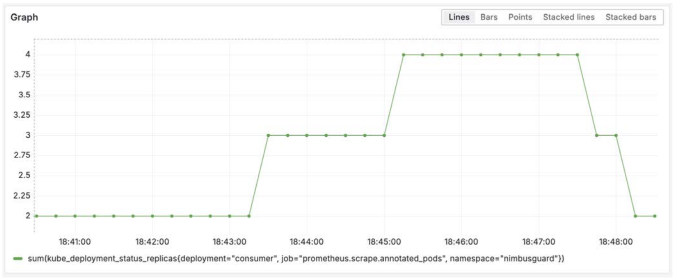

Fig. 2: HPA (Reactive Baseline)

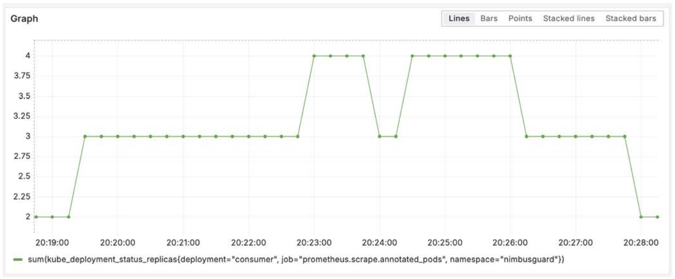

Fig. 3: KEDA (Flexible Trigger)

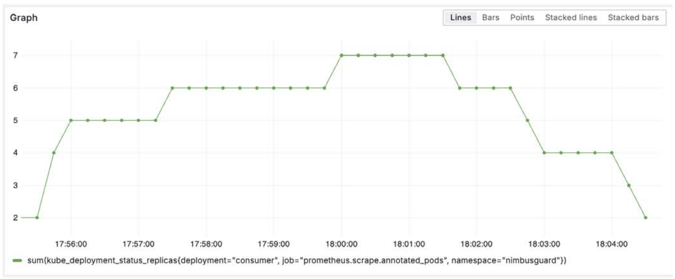

Fig. 4: NimbusGuard (Proactive)

TABLE II: Experimental Performance Comparison

Performance Metric	DQN	HPA	KEDA
Avg. Time to Scale (sec)	$\sim 60s \pm 5s$	$\sim 300s \pm 5s$	$\sim 90s \pm 5s$
Avg. Replicas (pods)	5.44	3.05	2.93
Peak Replicas (pods)	7	4	4
Total Scaling Events	8	4	4

DQN represents the proposed NimbusGuard system with Deep Q-Network intelligence with an uncertainty of $\pm 5s$

B. DQN-Specific Intelligence Analysis

A deeper analysis of the agent's internal state provides insight into its decision-making process. The efficacy of the primary proactive scaling feature, predicted memory utilization (s_1), is fundamentally dependent on the accuracy of the underlying forecasting model. An enhanced LSTM model was developed to predict memory usage, a key component of the state vector. The model was trained on historical data from the target application and evaluated on a hold-out test set.

Fig. 5 Analysis of the enhanced memory prediction model. The time series plot (left) shows a close tracking between actual and predicted values. The scatter plot (right) shows data points clustering tightly around the ideal fit line, visually confirming the high R² value.

Fig. 6 shows the immediate reward the agent received after each decision. The signal is a composite score derived from the multi-objective reward function, balancing performance, cost, and stability. The fluctuations reflect the constant trade-offs the agent must make. For example, a negative dip might correspond to a moment of temporary pod unavailability during a scale-up, which penalizes the agent and teaches it to scale more carefully in the future.

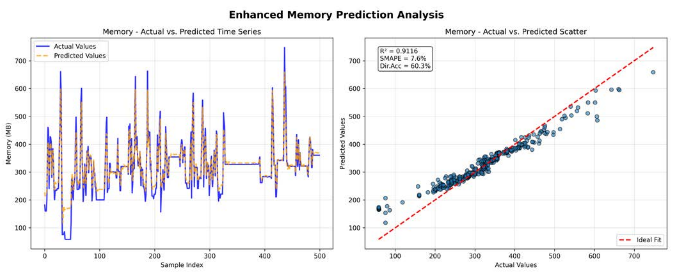

Fig. 5: LSTM Feature Analysis shows the proactive forecasts for load pressure and trend velocity.

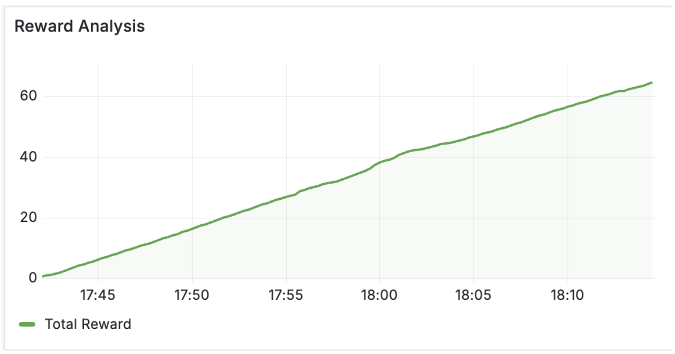

Fig. 6: Reward Analysis shows the evolution of the reward signal, which guides the agent's learning.

The LSTM forecasts (Fig. 5) provided the agent with proactive signals about future load pressure, enabling it to prepare for changes rather than just react to them. The divergence of these lines shows the agent learning to prefer actions that it predicts will lead to higher future rewards. The reward signal (Fig. 6) itself, while fluctuating, demonstrates the feedback loop that drives this learning process.

VI. Discussion

The results confirm NimbusGuard's algorithmic advantages; its multi-objective reward function and proactive LSTM forecasting enable it to anticipate load changes, a capability reactive systems like HPA and KEDA lack. However, the study is limited by the sequential evaluation of the autoscalers.

Future work should focus on two key areas:

- Expanding the action space (e.g., +2/-2 replicas) to allow for more aggressive responses to load spikes.
- Integrating Vertical Pod Autoscaling (VPA) to create a hybrid system that can choose between adding more pods or increasing the resources of existing ones.

References

[1] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.

[2] M. A. Tamiru, J. Tordsson, E. Elmroth, and G. Pierre, "An experimental evaluation of the kubernetes cluster autoscaler in the cloud," in *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2020, pp. 17–24.

[3] "Kubernetes," Jul. 2025. [Online]. Available: https://kubernetes.io/

[4] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the sixth ACM symposium on cloud computing*, 2015, pp. 167–167.

[5] R. Aurangzaib, W. Iqbal, M. Abdullah, F. Bukhari, F. Ullah, and A. Erradi, "Scalable containerized pipeline for real-time big data analytics," in *2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2022, pp. 25–32.

[6] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 958–972, 2021.

[7] "LangGraph — langchain.com," https://www.langchain.com/langgraph, [Accessed 30-07-2025].

[8] "GitHub - kedacore/keda: KEDA is a Kubernetes-based Event Driven Autoscaling component. It provides event driven scale for any container running in Kubernetes — github.com," https://github.com/kedacore/keda, [Accessed 28-07-2025].

[9] "KEDA — cncf.io," https://www.cncf.io/projects/keda/, [Accessed 28-07-2025].

[10] F. Rossi, V. Cardellini, and F. L. Presti, "Hierarchical scaling of microservices in kubernetes," in *2020 IEEE international conference on autonomic computing and self-organizing systems (ACSOS)*. IEEE, 2020, pp. 28–37.

[11] S. K. Mondal, X. Wu, H. M. D. Kabir, H.-N. Dai, K. Ni, H. Yuan, and T. Wang, "Toward optimal load prediction and customizable autoscaling scheme for kubernetes," *Mathematics*, vol. 11, no. 12, p. 2675, 2023.

[12] 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, p. 3835, 2021.

[13] A. A. Khaleq and I. Ra, "Intelligent autoscaling of microservices in the cloud for real-time applications," *IEEE access*, vol. 9, pp. 35 464–35 476, 2021.

[14] M. Imdoukh, I. Ahmad, and M. G. Alfailakawi, "Machine learning-based auto-scaling for containerized applications," *Neural Computing and Applications*, vol. 32, no. 13, pp. 9745–9760, 2020.

[15] M. Yan, X. Liang, Z. Lu, J. Wu, and W. Zhang, "Hansel: Adaptive horizontal scaling of microservices using bi-lstm," *Applied Soft Computing*, vol. 105, p. 107216, 2021.

[16] D.-D. Vu, M.-N. Tran, and Y. Kim, "Predictive hybrid autoscaling for containerized applications," *IEEE Access*, vol. 10, pp. 109 768–109 778, 2022.

[17] Y. Garí, D. A. Monge, and C. Mateos, "A q-learning approach for the autoscaling of scientific workflows in the cloud," *Future Generation Computer Systems*, vol. 127, pp. 168–180, 2022.

[18] Z. Jian, X. Xie, Y. Fang, Y. Jiang, Y. Lu, A. Dash, T. Li, and G. Wang, "Drs: A deep reinforcement learning enhanced kubernetes scheduler for microservice-based system," *Software: Practice and Experience*, vol. 54, no. 10, pp. 2102–2126, 2024.

[19] G. Zhang, W. Guo, Z. Tan, Q. Guan, and H. Jiang, "Kis-s: A gpu-aware kubernetes inference simulator with rl-based auto-scaling," *arXiv preprint arXiv:2507.07932*, 2025.

[20] S. Subramaniam and G. H. Loh, "Fire-and-forget: Load/store scheduling with no store queue at all," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 273–284.

[21] "asyncio Asynchronous I/O," https://docs.python.org/3/library/asyncio.html, [Accessed 30-07-2025].