DInos: A Deep Reinforcement Learning Approach to Generalizable Autoscaling in Stateless Cloud Applications

Constantinos Bitsakos[1](✉), Dimitrios Tsoumakos[2], Ioannis Konstantinou[3], and Nectarios Koziris[1]

[1] CSLAB, National Technical University of Athens (NTUA), Athens, Greece
{kbitsak,nkoziris}@cslab.ece.ntua.gr

[2] DBLAB, National Technical University of Athens (NTUA), Athens, Greece
dtsouma@mail.ntua.gr

[3] Department of Informatics and Telecommunications, University of Thessaly, Volos, Greece
ikons@uth.gr

Abstract. Efficient autoscaling in Kubernetes (K8s)-managed in-memory systems like Redis remains a critical challenge, especially under highly dynamic workloads. Traditional threshold-based mechanisms (e.g., HPA) often fail to anticipate sudden demand surges, leading to poor performance and inefficient resource use.

We introduce **DInos**, a Deep Reinforcement Learning (Deep RL) agent enhanced with LSTM layers and transfer learning, designed for proactive and adaptive autoscaling in Kubernetes. As an evolution of our earlier agent DERP, DInos leverages temporal workload modeling and pre-trained policies to generalize across deployments with minimal retraining. DInos utilizes a customizable reward function balancing throughput, latency, resource usage, and pod efficiency.

DInos achieves up to **17.3× higher rewards** in simulation and a **5.5× improvement** in real-world K8s-Redis deployments by forecasting spikes, optimizing pod counts and maintaining low latency, providing a robust autoscaling solution for volatile, cloud-native environments.

Keywords: Kubernetes · Deep Reinforcement Learning · LSTMs · Autoscaling Redis

1 Introduction

Kubernetes (K8s) is the de facto standard for orchestrating containerized applications, offering reactive autoscaling through mechanisms like the Horizontal Pod Autoscaler (HPA) [37]. However, such threshold-based approaches struggle with dynamic, time-sensitive workloads, often leading to overprovisioning or increased latency.

Redis, a stateless in-memory key-value database, is widely adopted in latency-critical cloud services [29]. Under traffic spikes, reactive autoscaling approaches frequently fail to respond well enough, resulting in degraded performance.

To overcome these limitations, our previous work introduced *DERP* [10], a Deep Reinforcement Learning (Deep RL) autoscaler. DERP significantly outperformed traditional threshold-based autoscalers and earlier RL-based autoscaling methods that used Q-tables or decision trees. In this paper, we further enhance DERP by introducing a new deep neural network architecture, extending it to utilize additional performance metrics such as CPU and memory usage, as well as pod count.

Building on these improvements, we propose our main contribution: **DInos**, an advanced LSTM-based Deep RL autoscaling agent that integrates transfer learning for rapid generalization across deployments [39]. DInos leverages temporal modeling and policy reuse to proactively anticipate workload changes and optimize resource allocation. Both DERP and DInos are trained with a customizable reward function that aims to balance throughput, latency, resource usage (CPU and memory), and pod efficiency [38], enabling versatile adaptation to diverse system goals.

DInos achieves up to a 17× improvement over DERP in simulation and a 5.5× gain over its non-transfer variant in real Kubernetes-Redis environments, clearly demonstrating the significant advantages of temporal modeling and policy transfer in proactive autoscaling. Furthermore, its transfer learning capability enables DInos to quickly and effectively adapt to diverse environments, new applications, different frameworks, and varying workloads without extensive retraining.

2 Related Work

Kubernetes (K8s) has emerged as a leading platform for cloud-native orchestration [4], driving substantial research in autoscaling. Traditional scaling mechanisms include the Horizontal Pod Autoscaler (HPA) [37], Vertical Pod Autoscaler (VPA) [6], and Cluster Autoscaler [3], relying on reactive CPU and memory thresholds. Such methods often struggle under rapidly changing workloads, causing resource inefficiencies and latency spikes.

KEDA [19], an event-driven Kubernetes autoscaler, improves flexibility using external metrics sources such as Kafka or Prometheus. However, its lack of predictive intelligence and adaptive learning limits its effectiveness with highly dynamic workloads.

Previous autoscalers employed reinforcement learning (RL) or heuristic-based approaches, notably Tiramola [34] and VCONF [28], initially targeting NoSQL or VM-based environments. Classic RL methods employing Q-tables or decision trees [24,27,36] had inherent scalability limitations and insufficient generalization.

Our earlier work, DERP [10], significantly outperformed these threshold-based and classical RL methods by employing Deep Q-Learning with neural networks. DERP introduced fine-grained autoscaling policies based on throughput, latency, and resource metrics. Despite these improvements, DERP lacked temporal modeling and knowledge transfer mechanisms, restricting its predictive accuracy and generalizability.

Other ML-based autoscalers, such as Imdoukh et al. [16] and the Bi-LSTM system by Dang-Quang and Yoo [11], enhanced workload forecasting but lacked an integrated adaptive RL framework, limiting responsiveness.

Research by Toka et al. [33] and Koukis et al. [18] explored predictive scaling and network overhead impacts, respectively. Studies like SCADIS [17], Harmonia [40], and performance benchmarks [9,22,30] provided insights into Redis-specific scenarios. Additional work investigated node-level elasticity [32], extreme-scale workloads [21], and dynamic cloud scaling strategies [7,25].

Advancements in LSTM network optimization [20,39], and Deep RL methodologies [15,38] underscored the importance of combining temporal forecasting with RL. Transfer learning has also been recognized as critical for rapid generalization across deployments [26]. Kubernetes-based ML management frameworks, like Scanflow-K8s [23] and performance studies by Fogli et al. [12], further emphasize the need for intelligent autoscaling.

Our proposed solution, **DInos**, significantly surpasses even the enhanced DERP architecture by integrating LSTM-based temporal forecasting, Double Deep Q-Learning, and transfer learning into a single agent. Unlike previous methods that treat forecasting and scaling separately, DInos learns proactive and predictive scaling strategies simultaneously. DInos demonstrates a substantial cumulative reward improvement of 17.3× over DERP in simulations and achieves a 5.5× reward gain compared to a non-transfer learning baseline in real Kubernetes/Redis deployments. These improvements result directly from DInos' capability to forecast resource needs, optimize provisioning proactively, and adapt rapidly to previously unseen workloads.

In summary, DInos represents an adaptive, intelligent autoscaling framework capable of leveraging historical workload patterns and generalizing across environments, establishing a robust solution for dynamic cloud-native autoscaling.

3 System Model and Background

3.1 Kubernetes and Redis Setup

Our system is deployed on a Kubernetes (K8s) cluster running a stateless Redis environment (Fig. 1). Redis, being an in-memory key-value store, is well-suited for dynamic autoscaling due to its statelessness and fast response times. Kubernetes orchestrates Redis pods and enables elastic scaling in response to workload fluctuations.

Metrics such as CPU-memory usage and latency are retrieved directly from the Kubernetes Metrics Server [5] and provided to our Deep RL agents for scaling decisions. Prometheus [2] and Grafana [1] were used only for monitoring and visualization. Our custom autoscaler interacts with the control plane to apply scaling actions via the kube-api-server.

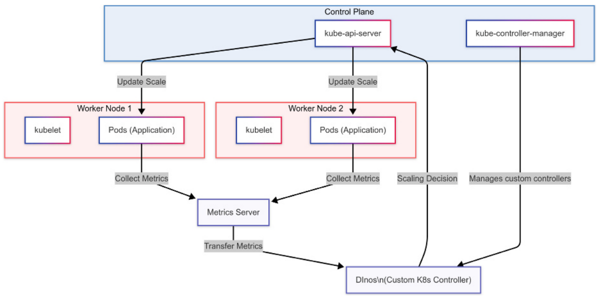


Fig. 1. System architecture of our proposed DInos autoscaling agent within a Kubernetes cluster. The agent interacts with the Kubernetes control plane by receiving time-series metrics from Redis pods via the Metrics Server. Based on these inputs, DInos issues proactive scaling decisions–adding or removing pods–by communicating with the kube-api-server. The agent learns workload patterns using deep reinforcement learning, enabling intelligent resource allocation that balances performance and cost across dynamic environments.

3.2 Deep Reinforcement Learning Agents

We compare two Deep RL-based agents: a dense-layer agent (enhanced DERP) and an LSTM agent enhanced with transfer learning (DInos). Both agents take a state vector of system metrics–load, CPU usage, memory usage, latency, and pod count–and return scaling actions.

DERP Agent (Dense). DERP enchance our previous work [10] by using two dense layers (128 units each) with ReLU activations. It is trained using a Double DQN [8] to avoid overestimation of Q-values and provides a baseline for comparison, This architecture is shown in Fig. 2.

– **Input:** System state vector
– **Hidden:** Two dense layers with 128 neurons
– **Output:** Q-values for scaling actions

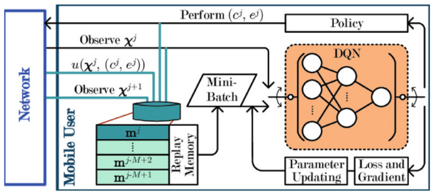


Fig. 2. Double Deep Q-Network (Double DQN) architecture used by both DERP and DInos for stable learning and decision-making. It separates action selection (online network) from value estimation (target network), mitigating Q-value overestimation and improving training stability. This architecture supports discrete action spaces, such as scaling decisions for pod count.

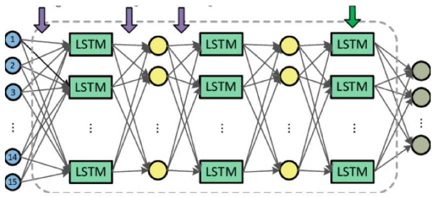


Fig. 3. LSTM-based neural network architecture employed by DInos. The LSTM layers are designed to capture long-term temporal dependencies in workload metrics, including periodic trends and abrupt spikes. This enables the agent to proactively scale resources by forecasting future system states rather than reacting to current load alone.

DInos Agent (LSTM + Transfer Learning). DInos extends the dense-layer-based DERP variant with LSTM layers and transfer learning. It leverages these LSTM layers to effectively model temporal patterns in workloads. Initially, a base model is extensively trained in a simulation environment whose parameters and load scenarios have been carefully formalized based on extensive real-world experiments and observations (Fig. 7, right). This pretrained model thus captures generalized scaling policies applicable to various workload patterns. DInos is subsequently initialized with this pretrained model when deployed in new environments, enabling it to rapidly adapt to different workloads with minimal additional training. Fine-tuning occurs on the target workload using a smaller batch size of 16, significantly accelerating adaptation and ensuring robust performance across diverse real-world deployment scenarios.

– **Input:** Sequence of past states
– **LSTM:** Memory layers with fine-tuning
– **Output:** Q-values for scaling decisions

The Spike Factor: While DERP reacts to general workload trends, it struggles with sudden spikes. DInos' LSTM layers anticipate these patterns using temporal memory, adjusting resources proactively–particularly effective for flash-sale or seasonal traffic surges (Figs. 3, 4).

3.3 Reward Function

The reward function balances five objectives:

$$\begin{aligned} \text{reward} = {} & \text{load\_factor} \cdot \text{next\_load} - \text{latency\_factor} \cdot \text{next\_latency} \\ & - \text{pods\_penalty} \cdot \text{num\_pods} - \text{cpu\_penalty} \cdot \text{cpu} - \text{memory\_penalty} \cdot \text{memory} \end{aligned} \tag{1}$$

This reward formulation is designed to highlight the versatility and capability of our agents in learning optimal scaling strategies under multiple objectives. While this particular reward function focuses on balancing throughput, latency, and resource consumption, it is fully customizable. System operators or cloud vendors can easily redefine the reward structure to reflect specific operational goals or application priorities–such as cost minimization, latency guarantees, or energy efficiency. DInos and enhanced DERP retain their effectiveness across such formulations, showcasing their adaptability to diverse autoscaling policies and deployment environments.

3.4 Simulation Environment

Workload patterns are generated using a sinusoidal base, Gaussian noise, and a sharp spike between timesteps 1500âĂŞ1600 to emulate bursty real-world traffic.

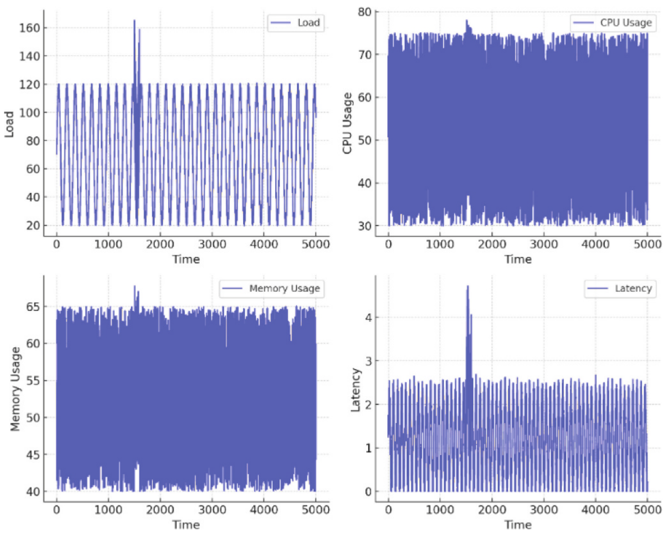


Fig. 4. Input metrics generated by the simulation environment: load, CPU usage, memory usage, and latency over 5000 timesteps. The load follows a sinusoidal pattern with added Gaussian noise and a sharp spike between timesteps 1500 and 1600, simulating a burst workload. These metrics form the state input for both enhanced DERP and DInos, testing their ability to maintain performance under dynamic and unpredictable conditions.

CPU, memory, and latency metrics are modeled accordingly after observing real-world behaviors (see Fig. 4 and the right subfigure of Fig. 7).

3.5 Training Details

Both agents are trained using the Huber loss function [14] for stability. A Double DQN framework (Fig. 2) is used to decouple action selection from value estimation and improve learning in noisy environments.

3.6 DInos Autoscaling Algorithm

Algorithm 1 DInos: Transfer Learning Autoscaler

```
 1: Input: Pretrained model (optional), current system state $s_t$
 2: Initialize Q-network (pretrained if using DInos)
 3: Initialize target network and replay buffer $R$
 4: for each timestep $t$ do
 5:     Observe $s_t$
 6:     if exploration condition met then
 7:         Select random action $a_t$
 8:     else
 9:         Select action $a_t = \arg\max_a Q(s_t, a)$
10:     end if
11:     Apply $a_t$, observe $r_t$, $s_{t+1}$
12:     Store $(s_t, a_t, r_t, s_{t+1})$ in $R$
13:     Sample minibatch from $R$
14:     for each $(s_t, a_t, r_t, s_{t+1})$ in minibatch do
15:         Compute target: $y = r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$
16:         if transfer learning is enabled then
17:             $Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha y$
18:         else
19:             $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta(y - Q(s_t, a_t))$
20:         end if
21:     end for
22:     Periodically update target network $\hat{Q}$
23: end for
```

4 Implementation

4.1 Training Setup

We trained two Deep RL agents–an enhanced version of DERP (with dense layers) and DInos (with LSTM and transfer learning)–using system metrics collected from a Redis cluster orchestrated by Kubernetes. The input state for each agent includes:

- Load (request rate to Redis),
- CPU usage,
- Memory usage,
- Latency,
- Number of pods.

The environment simulated dynamic workloads with oscillating patterns and abrupt traffic spikes (see Fig. 4), allowing agents to learn both reactive and proactive autoscaling behavior.

Agents selected one of three actions–add, remove, or maintain pods–every 15 s. The reward was computed at each timestep using the following weighted linear function:

$$\begin{aligned} \text{reward} = {} & 3.0 \cdot \text{next\_load} - 5.0 \cdot \text{next\_latency} \\ & - 10.0 \cdot \text{num\_pods} - 0.5 \cdot \text{next\_cpu\_usage} - 0.5 \cdot \text{next\_memory\_usage} \end{aligned} \tag{2}$$

This formulation promotes high system throughput (via the positive weight on `load`) and low latency, while discouraging over-provisioning and excessive resource usage. The specific weights were determined through empirical tuning and sensitivity analysis during simulation experiments. These coefficients are not fixed and can be adapted to suit different infrastructure priorities, such as cost minimization or latency-critical performance.

4.2 Kubernetes Integration

To enable real-time scaling by the agents, we disabled the default Horizontal Pod Autoscaler (HPA) and created a custom Python environment class that directly interfaces with the Kubernetes API. This class executes scaling actions issued by the agent and fetches current system metrics from the Kubernetes Metrics Server [5].

The cluster is configured with stateless Redis pods, ensuring that pod addition or removal can be handled gracefully without service disruption.

4.3 Challenges and Observations

Several technical challenges emerged during development:

- **Latency sensitivity:** Capturing realistic latency patterns required simulating spikes during load surges, guided by latency trends observed in the real-world environment. This helped ensure that the simulation environment reflected realistic system behavior and reinforced the agents ability to respond appropriately.
- **Temporal forecasting:** DERP, even the enchanced version, was unable to anticipate future workload shifts. In contrast, DInos successfully leveraged LSTM layers to recognize and prepare for recurring spikes, improving responsiveness (see Fig. 3).

- **Cost-performance tradeoff:** The pod penalty in the reward function helped balance throughput against overprovisioning, though it required careful tuning to avoid excessive scaling.

Despite these challenges, both agents were successfully trained in the simulated environment. DInos demonstrated superior performance by adapting to future workload conditions through transfer learning and temporal modeling. Full results are discussed in Sect. 5.

5 Experimental Results

5.1 Experimental Setup

Experiments were conducted on a Kubernetes cluster running on a large Ubuntu 20.04 VM with the following specs:

- **RAM:** 191 GiB, **CPU:** 32 vCPUs (Skylake, 2 GHz), **GPU:** Cirrus Logic, **Disk:** 750 GiB (EXT4)

Using Kubernetes on such a high-resource VM allowed the setup to emulate a full-scale cluster. We used Python 3.8 and PyTorch, and recorded load, latency, CPU-memory usage and pod count for evaluation.

5.2 Autoscaling Agents Compared

We evaluate two Deep RL-based autoscaling agents:

- **Enhanced DERP (Dense):** A Deep Q-learning agent using dense layers
- **DInos (LSTM + Transfer):** An advanced LSTM-based agent that leverages transfer learning for faster adaptation

5.3 Simulation Results

In the simulation environment our agents achieved:

- **DERP:** 2.2×10^5 reward
- **DInos:** 3.8×10^6 reward – a 17.3× improvement

Figure 4 illustrates the simulation environment, which features sinusoidal workloads combined with random noise and a sharp demand spike. Under these challenging conditions, DInos demonstrates a significant advantage over DERP, both in terms of accumulated rewards and system responsiveness. This superiority arises from DInos' effective forecasting and proactive scaling strategies. Detailed performance metrics and comparative analyses can also be found in Figs. 5 and 6. Specifically, Fig. 6 (right) clearly illustrates DInos' adaptive scaling capabilities, as it dynamically adjusts pod counts in response to workload fluctuations. The figure demonstrates how DInos efficiently balances resource usage and latency by proactively predicting workload patterns and swiftly responding to spikes, showcasing the adaptability and robustness of our agent under volatile simulated conditions.

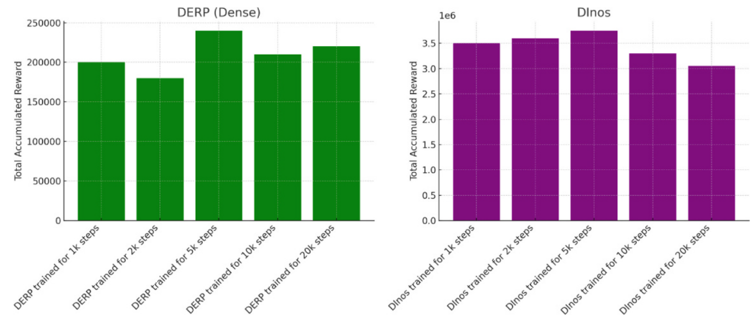


Fig. 5. Comparison of total accumulated rewards between DERP and DInos at different training checkpoints (1k, 2k, 5k, 10k, and 20k steps). While DERP's performance improves with training, it plateaus early. In contrast, DInos, leveraging LSTM layers and transfer learning, achieves significantly higher rewards across all training durations, highlighting its superior adaptability and learning efficiency in dynamic environments.

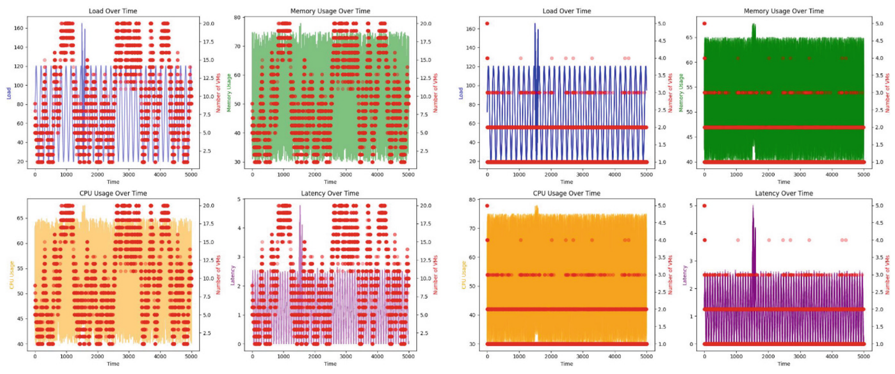


Fig. 6. Evaluation of DERP (left) and DInos (right) based on runtime metrics over 5000 timesteps. DERP responds reactively, often under- or over-provisioning during spikes. DInos, using LSTM layers and transfer learning, shows superior stability by anticipating load fluctuations and adapting pod count accordingly. The result is lower average latency, better CPU and memory efficiency, and higher throughput under varying load.

5.4 Real-World Results

In real Kubernetes/Redis deployments, DInos achieved:

- **DInos:** 308,000 reward
- **Baseline LSTM agent (no transfer):** 56,000 reward

In real Kubernetes/Redis deployments, DInos achieved a cumulative reward of 308,000, significantly outperforming its LSTM-based baseline without transfer learning, which attained a reward of only 56,000. Due to DERP's relatively lower performance in the simulation scenarios, we excluded it from real-world evaluations and instead focused on comparing DInos with its closest non-transfer-learning variant. This comparison demonstrates a remarkable 5.5× improvement, underscoring the practical benefits of incorporating transfer learning and temporal modeling into our RL framework.

Figure 7 (right) provides detailed metrics from the live Kubernetes/Redis deployment, highlighting the real-world adaptability of DInos. The figure explicitly demonstrates how DInos proactively manages resource provisioning by forecasting load variations and promptly scaling pod counts, maintaining stable resource utilization and consistently low latency despite noisy and spiky traffic. This confirms that DInos can rapidly adapt to diverse production workloads with minimal retraining effort, thereby validating the practical effectiveness and robustness of our proposed solution in dynamic cloud-native environments.

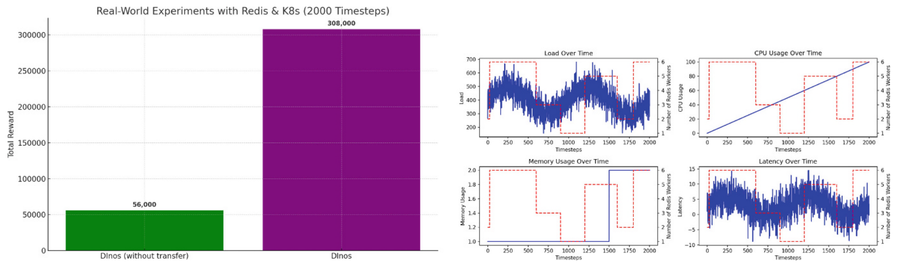


Fig. 7. Real-world evaluation of DInos in a live Kubernetes/Redis deployment. **Left:** Cumulative reward over 2000 timesteps, comparing DInos against its LSTM-only baseline. DInos achieves a total reward of 308,000, outperforming the LSTM-only version by 5.5×, showcasing the effectiveness of transfer learning in accelerating adaptation to production environments. **Right:** Temporal system behavior of DInos, demonstrating how the agent dynamically adjusts the number of Redis pods in response to noisy and spiky load patterns. The figure shows that DInos maintains low latency and stable resource usage while proactively scaling based on predicted workload trends.

5.5 Key Insights

- **Load Forecasting:** DInos anticipates spikes and scales preemptively via LSTM memory, unlike DERP which reacts post-factum.
- **Transfer Learning:** DInos requires minimal retraining and adapts quickly to new workloads, outperforming baseline LSTM agents in both accuracy and speed.
- **Resource Efficiency:** DInos minimizes pod count while maintaining low latency and high throughput.

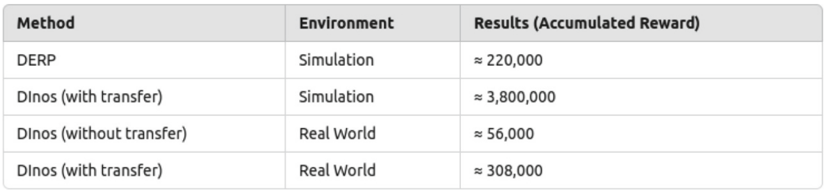
5.6 Performance Summary

Figure 8 highlights how **DInos** consistently outperforms the enchanced **DERP** agent across all key metrics–accumulated reward, CPU usage, memory usage, and system latency–in both simulated and real-world Kubernetes/Redis deployments. These metrics collectively demonstrate DInos' superior adaptability, forecasting ability, and efficiency under volatile workloads.

In simulation, DInos achieved a cumulative reward of 3.8×10^6, compared to just 2.2×10^5 for DERP, representing a **17.3× improvement**. This drastic gain reflects DInos' capacity to anticipate spikes and proactively allocate resources using its LSTM-enhanced decision-making. In real-world experiments, DInos outperformed a non-transfer LSTM baseline with a reward of 308,000 versus 56,000, demonstrating a **5.5× gain** in environments with real noise and system complexity.

Notably, these improvements are achieved while maintaining lower average CPU and memory consumption and significantly reducing latency fluctuations. This confirms that DInos does not simply overprovision resources to gain performance but instead learns to scale intelligently, minimizing cost while maximizing responsiveness.

The underlying reward function, which balances system throughput, latency, and resource usage, serves as a demonstration of the agent's capability to optimize under multi-objective conditions. However, this formulation is intentionally flexible: system operators or cloud vendors can customize the reward structure to reflect domain-specific policies–such as strict latency bounds, energy-aware scaling, or monetary budget constraints. Both DERP and DInos maintain robust performance across such variations, underscoring their **versatility and generalizability** in real-world autoscaling scenarios.



Method	Environment	Results (Accumulated Reward)
DERP	Simulation	≈ 220,000
DInos (with transfer)	Simulation	≈ 3,800,000
DInos (without transfer)	Real World	≈ 56,000
DInos (with transfer)	Real World	≈ 308,000

Fig. 8. Summary table comparing DERP and DInos across simulation and real-world deployments. Metrics include accumulated reward, average CPU and memory usage, and system latency. DInos consistently outperforms DERP in both environments, demonstrating better generalization, resource efficiency, and responsiveness to workload changes–especially in scenarios involving sharp demand spikes.

6 Discussion and Conclusion

Our experimental results confirm that Deep Reinforcement Learning (Deep RL) can outperform traditional autoscaling strategies in Kubernetes environments.

While DERP (dense-layer-based) improved over threshold-based baselines, it lacked the foresight needed for handling workload volatility. In contrast, DInos, our LSTM-based agent enchanced transfer learning, learned to proactively scale resources by identifying temporal workload patterns and leveraging knowledge from prior deployments.

In simulation, DInos achieved a 17.3× improvement in accumulated reward over DERP, and in real-world Redis deployments, it delivered 5.5× better performance compared to a baseline LSTM agent without transfer learning. These gains reflect more efficient resource utilization, lower latency, and better responsiveness under unpredictable workloads.

Future Directions

Future research can explore extending DInos to more complex environments:

– **Stateful Applications:** Applying DInos to stateful systems like Cassandra introduces new challenges such as data replication, consistency, and safe pod identity management due to the constraints of StatefulSets. Transfer learning can accelerate adaptation in such environments by leveraging prior knowledge to reduce retraining overhead.
– **Expanded Metrics:** Incorporating disk I/O, network bandwidth, or storage usage could provide a more holistic view of system health and enable even finer-grained scaling decisions.
– **Advanced RL Architectures:** Exploring algorithms like Dueling DQN [35], PPO [31], or SAC [13] may improve stability or allow continuous action spaces, especially in multi-objective environments.

Key Takeaways

DInos significantly improves upon prior Deep RL agents by combining LSTM forecasting with transfer learning. This enables faster convergence, better generalization, and minimal retraining across different deployments. Our approach demonstrates that predictive, learning-based autoscaling can lead to lower costs, higher performance, and robust adaptability–making it a compelling choice for modern, cloud-native systems.

These findings open new avenues for generalizing Deep RL-based autoscalers beyond Redis, across cloud platforms, and into more complex distributed workloads.

Competing Interests. The authors declare no competing interests relevant to this work.

References

1. Grafana: The open platform for analytics and monitoring (2021). https://grafana.com. Accessed 21 Oct 2024
2. Prometheus: Monitoring system and time series database (2021). https://prometheus.io. Accessed 21 Oct 2024
3. Cluster autoscaler (2024). https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/README.md. Accessed 22 Oct 2024
4. Horizontal pod autoscaler (2024). https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed 22 Oct 2024
5. Kubernetes metrics server (2024). https://github.com/kubernetes-sigs/metrics-server. Accessed 21 Oct 2024
6. Vertical pod autoscaler. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#vertical-pod-autoscaling (2024). Accessed 22 Oct 2024
7. Ardagna, C., et al.: A competitive scalability approach for cloud architectures. IEEE Trans. Serv. Comput. (2014). https://doi.org/10.1109/TSC.2014.2372786
8. Bell-Thomas, A.H.: Exploring variational deep Q networks. arXiv preprint arXiv:2004.05615 (2020). https://arxiv.org/abs/2004.05615
9. Ben Seghier, N., Kazar, O.: Performance benchmarking and comparison of NoSQL databases: Redis vs MongoDB vs Cassandra using YCSB tool. IEEE (2021). https://doi.org/10.1109/ICRAMI52622.2021.9585956
10. Bitsakos, C., Konstantinou, I., Koziris, N.: Derp: A deep reinforcement learning cloud system for elastic resource provisioning. In: 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 21–29. IEEE (2018)
11. Dang-Quang, N.M., Yoo, M.: Deep learning-based autoscaling using bidirectional long short-term memory for Kubernetes. Appl. Sci. **11**(9) (2021). https://doi.org/10.3390/app11093835
12. Fogli, M., et al.: Performance evaluation of Kubernetes distributions in federated cloud infrastructure. In: IEEE International Conference on Cloud Computing (2021). https://doi.org/10.1109/CLOUD.2021.00073
13. Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft actor-critic: off-policy maximum entropy Deep Reinforcement Learning with a stochastic actor. In: Proceedings of the 35th International Conference on Machine Learning (ICML), pp. 1861–1870. JMLR.org (2018)
14. Huber, P.J.: Robust estimation of a location parameter. Ann. Math. Stat. **35**(1), 73–101 (1964)
15. Ikemoto, J., et al.: Application of deep reinforcement learning to control problems. In: International Symposium on Control, Automation, and Systems (2019). https://doi.org/10.23919/ICCAS.2019.8912116
16. Imdoukh, M., Ahmad, I., Alfailakawi, M.G.: Machine learning-based auto-scaling for containerized applications. Neural Comput. Appl. **32**(13), 9745–9760 (2019). https://doi.org/10.1007/s00521-019-04507-z
17. Kimm, H., Li, Z., Kimm, H.: Scadis: supporting reliable scalability in Redis replication on demand. IEEE (2017). https://doi.org/10.1109/SmartCloud.2017.9
18. Koukis, G., Skaperas, S., Kapetanidou, I.A., Mamatas, L., Tsaoussidis, V.: Performance evaluation of kubernetes networking approaches across constraint edge environments (2024). https://arxiv.org/abs/2401.07674
19. Kubernetes-based Event Driven Autoscaling: KEDA: Kubernetes-based event driven autoscaling. https://keda.sh/ (2024). Accessed 31 Mar 2024

20. Kuchaiev, O., Ginsburg, B.: Factorization tricks for LSTM networks. arXiv preprint arXiv:1703.10722 (2017). https://arxiv.org/abs/1703.10722
21. Lankes, S.: Hermitcore: a unikernel for extreme scale computing. ACM (2016). https://doi.org/10.1145/2931088.2931093
22. Li, P., Luo, B., Zhu, W., Xu, H.: Cluster-based distributed dynamic cuckoo filter system for Redis. Taylor & Francis (2019). https://doi.org/10.1080/17445760.2019.1599889
23. Liu, P., et al.: Scanflow-k8s: Agent-based framework for autonomic management and supervision of ml workflows in Kubernetes clusters. In: International Conference on Machine Learning and Applications (2020). https://doi.org/10.1109/ICMLA.2020.00041
24. Lolos, K., Konstantinou, I., Kantere, V., Koziris, N.: Elastic management of cloud applications using adaptive reinforcement learning. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 203–212 (2017). https://api.semanticscholar.org/CorpusID:19567764
25. Malhotra, M., et al.: Dynamic scaling of web services for xen based virtual cloud environment. Int. J. Cloud Comput. (2020). https://doi.org/10.1504/IJCC.2020.10034234
26. Pan, S.J., Yang, Q.: A survey on transfer learning. IEEE Trans. Knowl. Data Eng. **22**(10), 1345–1359 (2010)
27. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, New York, NY, USA (1994)
28. Rao, J., Bu, X., Xu, C., Wang, L., Yin, G.: VCONF: a reinforcement learning approach to virtual machines auto-configuration. In: Proceedings of the 6th International Conference on Autonomic Computing (ICAC 2009), pp. 137–146. ACM (2009)
29. Sanfilippo, S., Stancliff, M.: Redis: A high-performance, in-memory, key-value store. Redis Labs (2013). https://redis.io/
30. Sanka, A.I., Chowdhury, M., Cheung, R.: Efficient high-performance FPGA-REDIS hybrid NoSQL caching system for blockchain scalability. Elsevier (2021). https://doi.org/10.1016/j.comcom.2021.01.017
31. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
32. Thurgood, B., Lennon, R.G.: Cloud computing with Kubernetes cluster elastic scaling. ACM (2019). https://doi.org/10.1145/3341325.3341995
33. Toka, L., Dobreff, G., Fodor, B., Sonkoly, B.: Machine learning-based scaling management for kubernetes edge clusters. IEEE Trans. Netw. Serv. Manage. **18**(1), 958–972 (2021). https://doi.org/10.1109/TNSM.2021.3052837
34. Tsoumakos, D., Konstantinou, I., Boumpouka, C., Sioutas, S., Koziris, N.: Automated, elastic resource provisioning for NoSQL clusters using TIRAMOLA. In: 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 34–41. IEEE (2013)
35. Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., de Freitas, N.: Dueling network architectures for Deep Reinforcement Learning. In: Proceedings of the 33rd International Conference on Machine Learning (ICML), pp. 1995–2003. JMLR.org (2016)
36. Watkins, C.J.C.H., Dayan, P.: Q-learning. Mach. Learn. **8**(3), 279–292 (1992)
37. Yeom, Y.J., Kim, T., Park, D.H., Kim, S.: Horizontal pod autoscaling in Kubernetes for elastic container orchestration. Sensors **20**(16), 4621 (2020)
38. Yu, K., Liu, Y., Wang, Q.: Review of deep reinforcement learning. IEEE Access (2020). https://doi.org/10.1109/ACCESS.2020.2979650

39. Yu, Y., Si, X., Hu, Z., Zhang, J.: A review of recurrent neural networks: LSTM cells and network architectures. Neural Comput. (2019). https://doi.org/10.1162/neco_a_01199
40. Zhu, H., Bai, Z., Li, J.: Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. ACM (2019). https://doi.org/10.14778/3368289.3368301