HPAQT: Adaptive and Interpretable High-level SLO-aware Autoscaling with Reinforcement Learning

Robin Mayerhofer
Distributed Systems Group
TU Wien
Vienna, Austria
mayerhofer1998@gmail.com

Andrea Morichetta
Distributed Systems Group
TU Wien
Vienna, Austria
a.morichetta@dsg.tuwien.ac.at

Alireza Furutanpey
Distributed Systems Group
TU Wien
Vienna, Austria
a.furutanpey@dsg.tuwien.ac.at

Schahram Dustdar
Distributed Systems Group
TU Wien
Vienna, Austria
dustdar@infosys.tuwien.ac.at

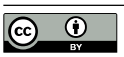
Abstract

Modern distributed applications rely on virtualized infrastructures to elastically meet their performance requirements. In this setting, autoscaling enables elastic adaptations at runtime. While allowing for the overcoming of the burden of adjusting the provisioned resources, autoscaling shifts the problem to the definition of an accurate and appropriate threshold, for example, a certain CPU usage, which is a difficult challenge to achieve. Furthermore, defining a priori a fixed value clashes with the dynamicity of modern applications and infrastructure, leading to inflexibility that can affect the quality of service over time. Finally, most autoscaling techniques rely on low, resource-level metrics, which, in complex scenarios, are difficult to gauge. In our paper, we propose HPAQT, a lightweight, stable, and reproducible RL mechanism that self-calibrates the autoscaling threshold to enforce composite, high-level objectives rather than fixed low-level metrics. HPAQT yields an easily interpretable, deployable, and effective policy. In experiments, HPAQT achieves 10× fewer violations than the reference Q-Threshold and beats the standard Kubernetes HPA, with less than 0.5% total violations in over 12 hours, thus demonstrating practical gains.

Keywords

reinforcement learning, auto-scaling, Q-Threshold, high-level SLO, workload, self-adaptive systems

ACM Reference Format:
Robin Mayerhofer, Andrea Morichetta, Alireza Furutanpey, and Schahram Dustdar. 2025. HPAQT: Adaptive and Interpretable High-level SLO-aware Autoscaling with Reinforcement Learning. In *2025 IEEE/ACM 18th International Conference on Utility and Cloud Computing (UCC '25), December 01–04, 2025, Nantes, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3773274.3774274



UCC '25, Nantes, France

ACM ISBN 979-8-4007-2285-1/25/12
https://doi.org/10.1145/3773274.3774274

1 Introduction

Virtualized platforms and environments have enabled the elasticity of distributed applications, letting them adjust the computing needs to meet quality of service requirements. In this context, autoscaling strategies help to dynamically allocate resources. Standard scaling approaches, as the Kubernetes Horizontal Pod Autoscaler (HPA) [1] rely on static thresholds to implement reactive strategies. First, while scaling removes the responsibility for the application's provider of assigning the infrastructure, it switches the burden to selecting an appropriate scaling threshold metric and target. Furthermore, the target might change together with the application fluctuations, making static thresholds unreliable for complex, dynamic applications. Overall, the caveat with common reactive autoscaling strategies is that they intrinsically result in over- or underprovisioning during sudden spikes. Consequently, misallocation results in substantial monetary losses for providers.

In recent years, Reinforcement Learning (RL) [2, 10, 22, 24, 27, 30, 31] emerged as an answer to the static nature of standard autoscalers, pushing towards (self-)adaptive solutions. However, despite a decade of considerable research, no large providers have applied RL-based autoscalers. We argue that two underlying, but related limitations of current RL methods prevent them from being productively used. First is the lack of *interpretability* from decisions. Even if providers can reduce wasting resources from a seemingly reliable autoscaler, they risk unpredictable failure, leading to prolonged periods of Service Level Objectives (SLOs) violations or, worse, near-complete outages. For example, if an autoscaler has complete control over the number of replicas, it may starve services except one it overwhelmingly favors. Hence, providers need to trust the autoscaler and understand decisions so that careful monitoring can yield insights about its behavior [10]. Second, is the trade-off between domain generalization and performance, particularly for autoscalers based on the popular Q-learning method [27]. Methods that tightly link high-level SLOs or applications to their autoscaling algorithm work well for their intended domain but require significant effort to adapt them to different domains. Conversely, methods that frame the objective of an autoscaling agent to generalize across a wide range of domains may perform below the expectations of

[1] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

providers [27]. We argue that the cause of both limitations is the action- and state-space explosion leading to extended learning and improvement phases [2], [10]. This last shortcoming is in general evident with Deep Reinforcement Learning methods, where the use of deep learning structures can lead to long and complex training, and result in overfitting [12]. While this may be an intrinsic limitation of RL learning methods that require complex sequences of actions, we argue that a large action space is unnecessary for an autoscaling agent to act on their reasoning about potential outcomes. Furthermore, the democratization and access to distributed resources have led the request to focus on offering higher-level management tools to the application providers [7, 17]. They have expectations in terms of how their applications are intended to perform from a high-level, business-level perspective, so, in the same way, autoscalers must optimize what users and businesses actually care about (latency, errors, cost, energy) rather than lower-level metrics. In this direction, components that allow the management through high-level, SLO-aligned metrics [23] are a necessary requirement. Still, properly making sure that these high-level SLOs translate into actionable inputs for infrastructure management is tricky, as business-to-resource relations are not easy to derive.

Our goal is to overcome the limitations of static thresholds and heavy DL-RL, which make autoscaling either fragile or opaque. In this direction, we present a refreshingly simple RL-based autoscaler from which interpretability and adaptivity arise from its drastically reduced action space, and make its implementation *publicly available*. [2] We propose both a double-threshold and a single-threshold autoscaling policy. Our implementation builds on the Q-Threshold approach proposed by Horovitz et al. [10], with a focus on the fulfillment of high-level requirements. The goal is to have a lightweight, generalizable, and interoperable RL mechanism that, by only changing the threshold for the autoscaling agent's objective, can trigger better-scaling actions to avoid intent violations. In particular, we improve on Q-Threshold by proposing a series of adaptations, which involve reducing the state space, keeping the Q-Table values over time, and adopting SLO-aware strategies. Leveraging Q-learning based approaches, we guarantee the interpretability of the policy, as action-values per state are explicitly coupled. This approach is also stable and reproducible as it requires few implementation knobs. Furthermore, we let our proposed approach generalize over both low- and high-level SLO metrics for the reward calculation, creating an effective bridge between the providers QoS requirements and the actuation of strategies on the infrastructure.

The evaluation, performed on a realistic use case of image classification in a dynamic application with real-time guarantee requirements, shows that our solution can outperform the existing Q-Threshold approach and Kubernetes HPA. In particular, using request duration as a SLO, HPAQT produces 10 times less violations than Q-Threshold, and improves also on the widely-used HPA autoscaler. Furthermore, HPAQT shows to work in even with the composite, complex cost-efficiency SLO. We allow a fair comparison with static, reactive rules, avoiding pretraining for the RL agents and letting them learn at runtime. The positive results provided by HPAQT show its potential as a scalable, fast approach.

[2] Our approach: https://github.com/robinmayerhofer/polaris-reinforcement-learning/tree/main

In summary, our main contributions are:

- **High-level SLO–aware autoscaling via a lightweight RL threshold tuner (HPAQT)**: this method allows to map composite, complex goals (e.g., latency SLOs, cost-efficiency) to adaptive thresholds, avoiding manual tuning and limits of standard reactive scaling strategies.
- **Stability and interpretability adaptations to Q-Threshold**: by simplifying the state space and introducing the Q-Table Memory for knowledge retention, and adjusting the learning strategy, we offer a lightweight, yet stable and reproducible approach to autoscaling.
- **Robust evaluation in a realistic Kubernetes and TensorFlow-Serving scenario**: the evaluation highlights that HPAQT leads to fewer SLO violations when compared to the reference Q-Threshold RL baseline and to Kubernetes HPA; furthermore, we include a cost-efficiency objective study showing how HPAQT can deal with composite SLOs.

The remainder of this paper is structured as follows. Section 2 reviews related work in RL- and threshold-based autoscaling. Section 3 formulates the problem, identifying key limitations in existing solutions. Section 4 details the proposed approach, focusing on adaptations to Q-Threshold, including single-threshold models, memory retention, and update strategies. Section 5 presents an extensive evaluation comparing the proposed adaptations against baselines, showing that the best configuration achieves up to 10x fewer SLO violations. Section 6 concludes with future research directions for high-level SLO-aware autoscaling.

2 Related Work

Serverless platforms. Schuler et al. [28] investigate the applicability of Q-learning for request-based autoscaling in Knative, demonstrating that RL can dynamically determine concurrency limits to improve throughput and latency without prior workload knowledge. Xue et al. [32] propose a meta RL-based predictive autoscaling strategy that incorporates deep periodic workload prediction and Neural Process models to enhance decision-making, achieving significant performance improvements when deployed in Alipay's cloud infrastructure. Agarwal et al. [1] explore recurrent reinforcement learning (LSTM-PPO) for autoscaling in Function-as-a-Service (FaaS) environments, modeling scaling decisions as a Partially Observable Markov Decision Process (POMDP) and showing an 18% improvement in throughput over standard PPO-based policies. Zhang et al. [33] introduce a Q-learning-based adaptive autoscaling method that combines horizontal and vertical scaling to optimize resource allocation while meeting strict latency constraints, reducing costs by 10.5% on average compared to state-of-the-art approaches. These works collectively highlight the growing role of RL in intelligent autoscaling for serverless applications, enabling efficient, adaptive, and QoS-aware resource management in dynamic cloud environments.

Cloud platforms. Qiu et al. [25] introduce AWARE, an RL-based framework for workload autoscaling in production cloud systems, leveraging meta-learning and bootstrapping to enhance adaptation speed and robustness, achieving a 5.5× faster policy adaptation and reducing SLO violations by 16.9×. Xue et al. [32] propose a meta RL-based predictive autoscaling strategy incorporating deep

periodic workload forecasting and Neural Process models, which outperforms existing methods in accuracy and has been successfully deployed at Alipay to handle large-scale cloud workloads. Liu et al. [16] design a meta-gradient RL-based scheduling framework for time-critical tasks in cloud environments, significantly improving robustness and adaptation speed while ensuring deadline guarantees under dynamic workloads. Chrysopoulos et al. [5] introduce RBS-CQL, a Deep RL system integrating offline learning for cloud elasticity, demonstrating a 10% improvement in autoscaling efficiency compared to online training for Kubernetes-based NoSQL applications. Arabnejad et al. [2] instead focused on the RL model performance, using fuzzification to reduce the state space (fuzzy workload and fuzzy response time), together with a smaller action space. Mishra et al. [18] develop a Q-learning-based RL autoscaler that dynamically scales Kubernetes workloads beyond the limitations of HPA, providing customized resource allocation based on throughput, latency, and CPU utilization. Horovitz et al. [10] identified the issues when scaling approaches to larger environments and proposed a simple, but effective solution called Q-Threshold to control only the target for autoscaling and not directly the scaling actions. Similarly, other solutions focused on offering RL algorithms to manage autoscaling thresholds [4, 8, 11], but typically they focus on low-level SLOs and metrics. More similarly to us, Rossi et al. [26] offered an evolution of Horovitz's work. Different from our approach, though, they focus more on controlling multiple metric thresholds rather than adjusting them based on high-level SLOs. Furthermore, their focus is on CPU and memory as scaling metrics.

2.1 Takeaways

Compared to the existing RL methods, our HPAQT approach presents some key differences. Compared to RL-based autoscaling for serverless and cloud platforms, we leverage a unique state space design tailored for dynamic workload adaptation, optimizing decision-making without relying on workload periodicity assumptions, as for example in meta-RL and LSTM-based models. Additionally, while existing RL autoscalers often use predefined scaling constraints or limited action spaces, our approach has a more adaptive action space, thanks to the threshold manipulation. This design allows for autonomous, real-time adaptation without requiring explicit dependency models, making it more flexible for heterogeneous cloud environments. Compared to other threshold-manipulation methods, HPAQT stands out for its simple yet domain-accurate approach. By relying on a minimal state space, adding a memory for the Q-Table, and employing a simple heuristic, we offer a precise and reliable tool for dynamic autoscaling.

3 Problem Formulation

Auto-scaling in cloud environments requires dynamically adjusting resources to meet fluctuating workload demands while maintaining performance guarantees. Traditional autoscalers rely on *static threshold values* (e.g., CPU utilization) to trigger scaling decisions. While these approaches can adequately meet demand in expected scenarios, they fail in dynamic workloads where the scheduler may over- or under-provision resources. In both cases, the cloud provider incurs losses—either by wasting resources or risking SLA breaches. Conversely, RL-based autoscalers aim to reduce operational costs and improve service quality by accurately predicting demand. Decades of research have produced methods that perform well on benchmarks (2). However, there is currently no evidence that they see significant real-world adoption in major cloud platforms. This lack of adoption stems from a critical limitation: operators cannot reliably predict or control RL scheduler behavior in complex production systems. The unpredictability of RL-based systems presents an unacceptable operational risk, as unexpected scaling decisions can trigger cascading failures and widespread outages. In other words, insufficient interpretability categorically disqualifies RL-based autoscalers from real-world deployment at scale. We systematically evaluate different **RL adaptations** (e.g., memory retention, threshold update strategies) and investigate their impact on **stability, interpretability, and performance**. We conduct a comprehensive comparison against standard **HPA-based** autoscaling and the **original Q-Threshold model**. Our evaluation shows that the standard Q-Threshold, despite its theoretical flexibility, suffers from the worst performance due to instability and inefficient scaling decisions. By contrast, our single-threshold adaptation significantly improves intent compliance and stability, demonstrating the practical advantages of simplifying RL-based autoscaling. By bridging the gap between **high-level intent-based cloud management** and **practical autoscaling**, this work aims to provide an **interpretable, adaptable, and production-ready** reinforcement learning-based autoscaler.

4 Approach

We approach building a self-adaptive autoscaling policy by adapting and augmenting the existing Q-Threshold method [10]. Based on our adaptation, we implement two approaches, the first, *Augmented Q-Threshold*, keeps the original lower- and upper-threshold structure. In the second, HPA Q-Threshold, we instead follow the principle of Kubernetes HPA by simplifying the policy to a single threshold.

4.1 Original Q-Threshold

Q-Threshold is aRL-based autoscaling technique that aims to fix or improve early-stage performance, issues introduced by convergence speed, and worst-case performance [10]. Moreover, Q-Threshold aims to be a solution that industry experts can trust because of its (perceived) simplicity and similarity to the industry-standard threshold-based autoscaling, thus reducing risk.

Environment Modeling. Q-Threshold aims to keep small state/action spaces, because of issues identified when at least one of them gets big. The state space contains the *current number of resources*. The action space for the agent corresponds to increasing/decreasing the threshold $(-2, -1, 0, +1, +2)$. Thus, the Q-table is very simple, as shown in 1. When an action changes the threshold, then the values in the Q-table are shifted. For example, if the action +2 is chosen, then the Q-values for the current state (before applying the action) are shifted right twice, and two 0 values are introduced for +1 and +2. Additionally, the agent maps each state to a threshold used if the agent is in the given state, as shown in 2.

For the reward, they want to ensure that there are no Service Level Agreement (SLA) violations which occur when exceeding the

# Resources / Action	-2	-1	0	1	+2
3	0	0	3.92	-12.5	-12.6
4	0	0	2.03	3.68	1.93
5	0	0.15	4.29	0.41	-0.15
...					

Table 1: Q-table for Q-Threshold. Taken and adapted Table VI (left) from [10].

# Resources	Threshold
3	80%
4	72%
5	77%
...	...

Table 2: Mapping of the state to the threshold. Taken and adapted Table VI (right) from [10].

configured 95^{th} percentile response time:

$$r = \begin{cases} \frac{1-e^{-p(1-\frac{respTime}{SLA})}}{1-\rho}, \text{if } respTime > SLO \\ \frac{1-e^{-p}}{1-\rho}, \text{if } respTime < SLO \end{cases} \quad (1)$$

Here, ρ is the upper-/lower-threshold value, and p is a parameter determining the steepness of the exponential [10]. As p is set to a fixed value, the reward is essentially influenced by the SLO compliance level and the threshold. As long as the SLO is met, the reward is positive and only depends on the threshold (higher thresholds lead to higher rewards). If the SLO is violated, the reward is negative. Moreover, the reward strictly monotonically decreases based on the SLO compliance level and is amplified by the threshold (higher threshold, bigger negative reward).

Concept. The autoscaler manages two thresholds (and thus, also *two agents*). If the lower one is passed, scaling-in happens, and the higher one determines when scaling-out happens. Each agent tries to adapt their thresholds to optimize the received reward. Horovitz and Arian also proposed an algorithm determining which Q-table (for lower/upper threshold) should be updated, Moreover, for exploration, an ε-greedy policy is chosen that is biased towards higher thresholds, i.e., more exploration towards higher thresholds, which results in scaling-in as early as possible and scaling-out as late as possible.

4.2 Augmented Q-Threshold

We re-implement and extend the Q-Threshold method by introducing four adaptations to improve scalability, generalization, and stability: the **Generalizable Reward** function, the **Single State (1STATE)** simplification, **Q-Table Memory (MEMORY)**, and the **Least-Extreme Then Random (LE-STRATEGY)** action selection policy. Here, we still maintain the Q-Threshold structure, depicted in Fig. 1, and consisting of two agents, with their own Q-tables, locally managing lower- and upper-thresholds.

4.2.1 Generalizable Reward. First, we want to generalize the Q-Threshold algorithm to enable providing a different reward. The original Q-Threshold algorithm relies on a reward function that

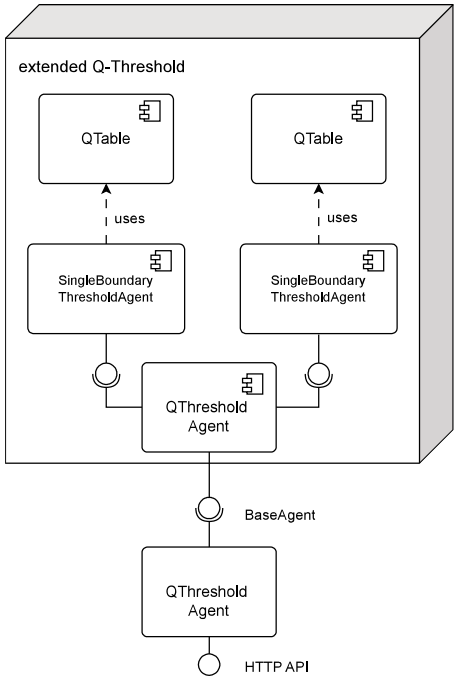


Figure 1: Augmented Q-Threshold components.

hypothesizes the need to keep the target metric within certain boundaries. This use case is typical when handling resource metrics as CPU or memory, where lower and upper boundaries indicate the need for scale-in (or -down) and scale-out (or -up), respectively. This structure, however, limits the metrics the algorithm can work with. When dealing with high-level SLOs, though, we could face the instance of a more ambiguous metric. As an example, we can consider cost efficiency, which is typically expressed [9, 15] as:

$$\text{Cost Efficiency} = \frac{\text{Successful requests per second within a threshold}}{\text{Total workload cost}}$$

Imagine the system is experiencing high load. The RL agent observes a drop in cost efficiency. This decline could be either because too few resources are allocated, meaning that requests are dropped (under-provisioning), or too many resources are used without improving performance (over-provisioning). However, based on cost efficiency alone, the agent can't tell which case it is. It may make the wrong decision, for example, increasing the threshold when it should be lowering it to trigger earlier scaling. We solve this by implementing a modular component to plug in the reward. For example, in the case of cost efficiency, we let the user define their boundaries. This step is essential also because it enables using (high-level) SLOs to influence the decisions of the RL agent.

4.2.2 Single State (1STATE). In the original Q-Threshold approach [10], the agent maintains a Q-table indexed by both the current replica count and the scaling threshold. This allows the agent to learn replica-count–specific threshold adaptation policies: for instance, to scale in more aggressively when many replicas are active and more conservatively when only a few remain. While effective, this results in a larger state space and sharp changes in

threshold when the replica count changes rapidly. Furthermore, it requires maintaining an auxiliary data structure that maps each replica count to its current threshold. The **1STATE** adaptation removes this dependency by reducing the Q-table to a *single row* and discarding the use of replica count as state information. The agent operates with a single, constant dummy state and learns a general policy for threshold adaptation, independent of the current scale. Internally, this is implemented by storing and retrieving the threshold using a fixed state index, e.g., `replica_count_to_threshold[0]`. This simplification offers three potential benefits: (i) reduced memory and implementation complexity, (ii) faster learning due to fewer state-action pairs, and (iii) generalization across scales. However, it also removes the agent's ability to reason about the impact of scaling decisions relative to system size. This trade-off is evaluated empirically in Section 5.

4.2.3 Q-Table Memory (MEMORY). In the standard Q-Threshold implementation, threshold adaptation is implemented as a shift over the Q-table. For example, applying action +2 moves the threshold up by two units, shifting the Q-values accordingly. Q-values that fall outside the new window are discarded and replaced with zeros. This can lead to a loss of accumulated knowledge and introduce instability during training, mainly when threshold changes occur frequently. To address this, we introduce a **Q-Table Memory** mechanism. Instead of discarding values when the threshold shifts, we treat the Q-table as a sliding window into a larger, persistent memory. Q-values that move outside the current view are retained, and if they re-enter later, their previous values are restored. This allows the agent to accumulate and reuse long-term knowledge, stabilizing learning and improving convergence.

4.2.4 LE-STRATEGY: Least-Extreme Then Random. In reinforcement learning, multiple actions can sometimes have identical Q-values. The original Q-Threshold method does not specify how to resolve such ties. To introduce a consistent and conservative resolution strategy, we implement the **Least-Extreme Then Random (LE-STRATEGY)** policy. When multiple actions share the maximum Q-value, the agent selects the one that causes the slightest change to the threshold (i.e., the action closest to zero). If multiple such minimally disruptive actions exist (e.g., -1 and $+1$), one is selected at random. This approach reduces the likelihood of oscillatory behavior and leads to smoother, more stable threshold adjustments.

4.3 HPA-Q-Threshold (HPAQT)

Q-Threshold is designed to work with two thresholds, scale out when the metric is above the upper one, and scale in when the metric is below the lower one. This is how many traditional autoscalers work. However, industry-standard autoscalers for containers such as HPA from Kubernetes use a single threshold - they maintain a desired metric value. This allows to scale out or in faster because multiple instances could be added/removed at the same time. Moreover, it is more straightforward as it only requires maintaining a single Q-table instead of two. Thus, we want to evaluate both Q-Threshold with a single threshold and with two thresholds. Fig. 2 shows the components that build HPAQT. Furthermore, in Alg. 1 we highlight the main steps. First, the RL agent monitors the current state, then, based on its current strategy, takes an action. If this action triggers the change of a threshold's value, then HPA checks whether this leads to an autoscaling. Eventually, the number of replicas will be adjusted accordingly. Finally, HPAQT updates the Q-Table memory.

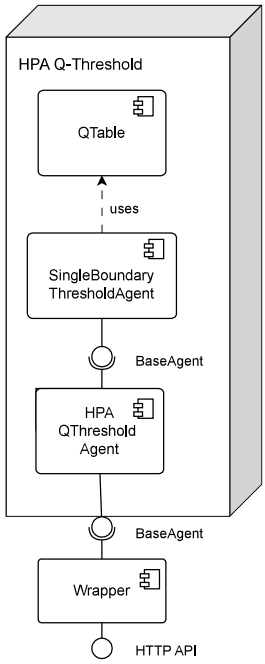


Figure 2: HPA-Q-Threshold components.

4.4 Summary

We design a set of adaptations for Q-Threshold autoscaling that are (i) **application-agnostic**, i.e., adapting to an ever-changing landscape, where each service has different requirements and SLOs to fulfill. Furthermore, (ii) we want to allow **high-level SLOs** to be used to scale services. The mechanism also has to be (iii) **interpretable**, as the autoscaling solution needs to be trustworthy. In addition, the approach should be (iv) **easily configurable** and provide an excellent autoscaling policy. Our solution should require (v) **no additional sandboxed environments** to evaluate an application before deploying it, as it would represent a slowdown. As updates and up- and downstream dependencies can influence the performance of an application, an autoscaling solution must be able to (vi) **react to such changes** and be able to update its policy when needed to ensure appropriate performance. All of these requirements are related, and it is possible to (vii) **make trade-offs** for one or the other. If a solution already has excellent worst-case performance, then early-stage performance, by definition, is also good, and slower convergence is a minor issue. On the other hand, (extremely) fast convergence helps to neglect the bad early-stage and worst-case performance issues.

Algorithm 1: HPA-Q-Threshold (HPAQT): General Control Loop with 1STATE, MEMORY, LE-STRATEGY

Input: Discrete threshold indices $\mathcal{T} = \{0, \ldots, T_{\max}\}$; actions $\mathcal{A} = \{-2, -1, 0, +1, +2\}$; replica bounds $[N_{\min}, N_{\max}]$
Input: Learning rate α, discount γ, exploration ε with decay; controlled metric m; SLO definition
Data: Q-Table Memory (MEMORY): $Q[0..T_{\max}][a \in \mathcal{A}] \leftarrow 0$ (persistent across threshold shifts)

Initialize threshold index $t \leftarrow t_{\text{init}}$ (1STATE); $\varepsilon \leftarrow \varepsilon_{\text{start}}$.
while *system running* **do**
  // 1) Observe (1STATE)
  obs $\leftarrow$ ReadMonitoring(); $x \leftarrow$ obs$[m]$; $n \leftarrow$ current replicas
  // 2) Action selection with LE-STRATEGY
  $a \leftarrow \text{LESelect}(Q[t][\cdot], \mathcal{A}, \varepsilon)$
  // 3) Threshold index update
  $t' \leftarrow a$
  // 4) Index $\rightarrow$ target and HPA actuation
  $n^{\star} \leftarrow$ HPAComputeDesiredReplicas$(t', n, N_{\min}, N_{\max})$;
  // 5) Next observation and SLO-aware reward
  obs$'$ $\leftarrow$ ReadMonitoring(); $r \leftarrow$ Reward(obs$'$, SLO)
  // 6) Q-update in MEMORY (tabular Q-learning, 1STATE)
  $Q[t][a] \leftarrow shift$

5 Evaluation

In this section, we evaluate our Reinforcement Learning (RL) autoscaling solution by setting up a comprehensive testbed and executing a series of experiments against established baselines using a specific use case.

5.1 Evaluation Testbed Setup

We evaluate our RL autoscaling solution using TensorFlow Serving (ModelNetV2) for image classification in a Kubernetes environment. This use case is essential as many modern applications might need to classify images to make crucial decisions at runtime. One relevant use case is disaster rescue [3, 6, 29]. Here, we envision that the application owner executes the logic for detecting if, in emergencies, flying drones have detected people or human-owned equipment. In this scenario, it is fair to believe that the stakeholder would ask for performance guarantees, where they want the drones to receive a label for the images they capture from a model served through a real-time cloud platform. A real-time constraint can be translated into a high-level objective of *500ms request duration* [14] (using the 95th percentile information that we collect from the monitoring tools), i.e., guaranteeing that the system would process two frames per second, which is going to be our main target. In Fig. 3, we depict how our strategies are applied to orchestrate the containers. Each experiment ran for twelve hours with varying load patterns to allow the RL agent to learn scaling policies. The testbed included a 22-core, 32GB RAM VM with MicroK8s, Prometheus, NGINX, and Kubecost for metrics collection and cost monitoring.

5.2 Request duration SLO

Here, we compare HPAQT with the baseline methods on the target SLO of request duration and use CPU as a scaling metric. The reward is calculated on the SLO, but the threshold is based on the CPU usage values. We use CPU usage as a target, as it is typically used in related works. To this end, we aim to offer a comprehensive comparison, measuring the difference between the Kubernetes HPA and the Q-Threshold baseline against our adapted formats.

The first dimension for the evaluation is the request duration SLO. In the tests summarized by Fig. 4 and Fig. 5, we compare how much the autoscaling approaches help reduce the number of violations and how overall the *request duration* values are kept within acceptable boundaries. Our proposed HPAQT algorithm (in red) widely outperforms all the other approaches. Considering the violation (see Fig. 4), HPAQT fails to fulfill the SLO target only 0.49% of the time, more than half of the time compared to the second best, HPA (in blue), which generates 1.04% of violations. The impact of our augmentations is also visible when comparing the augmented Q-Threshold (in green) with its baseline model. The difference is stark, with 6.83% of violations with our proposed improvements, versus 31.51% without, almost five times less. This difference is also highlighted by the boxplots in Fig. 5. There, we can see how HPAQT keeps the good values for the request duration. In particular, at the bottom of Fig. 6, we can see that, especially for HPA and HPAQT, the most significant magnitude in the SLO violation is in the beginning. The explanation is that the first request to a new TensorFlow serving pod takes more than 2.5 seconds, while subsequent requests can generally be much faster.

We dive into the various policy performances by inspecting the number of pods generated over time and the CPU usage. Fig. 7 shows that HPAQT, starting with a lower threshold, guarantees a steady performance, with only a few CPU usage peaks above 50%; HPA is also balanced. Interesting is the behavior for the Q-Threshold implementations; the *baseline* has an overall intense CPU usage, which likely leads to the large number of violations. Instead, the *augmented* Q-Threshold shows high variability, with extreme peaks and lows. Fig. 8 allows us to examine how the approaches scale the workload. We can see how the baseline and augmented Q-Threshold are more conservative; they scale less often and to a smaller extent. Conversely, HPA and the proposed HPAQT show swift variations, with HPAQT scaling up to 10 replicas. These autoscaling performance differences clearly impact the overall SLO fulfillment, as discussed above.

5.3 Cost efficiency high-level SLO

Here, we perform a test where we consider an even higher level SLO as a reward for the RL agent. In particular, as we discussed in Sec. 4, we only have a lower boundary in this case. Therefore, any action that can increase the cost efficiency shouldn't be penalized. For this reason, we show how our single threshold HPAQT approach performs. To do so, we introduce a cost efficiency target. We set the target equal to 10, implying that at least one pod should, on average, process at least two requests per second. Of course, the

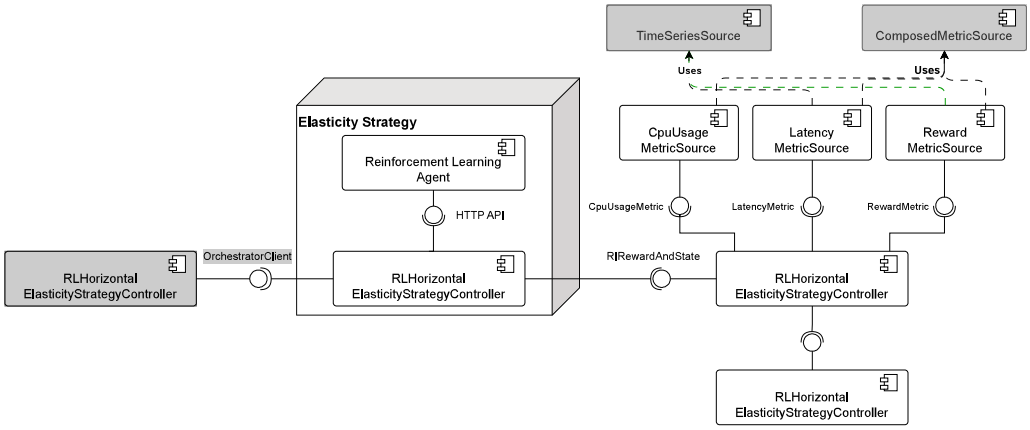


Figure 3: Architecture of our elasticity controller.

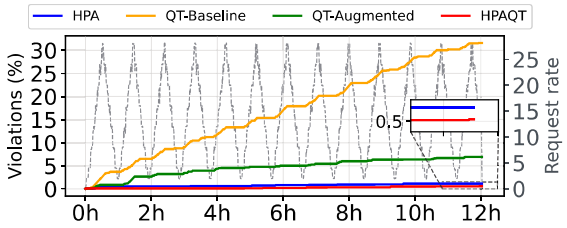


Figure 4: Violations over time for the four evaluated algorithms.

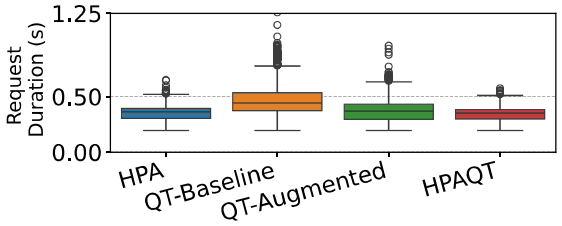


Figure 5: Boxplots of request duration values across the observation period.

more the load is supported, the better. Furthermore, we introduce a different scaling metric that the agent can control, i.e., by directly limiting the *request duration*. We compare it with the control on *CPU usage*. As we can see, the number of violations for HPAQT is overall higher than when targeting the request duration. Still, the violation rate is a maximum of 2.5% every two hours, which is still sustainable and can be adjusted by the service owner. At the same time, we can notice the difficulty of scaling by tuning the threshold of a higher-level target, such as request duration.

5.4 Discussion

While HPAQT has shown promising results, more effort is necessary to enhance the robustness of our solution. One of the weak aspects highlighted in the evaluation concerns the least-extreme strategy. While this approach guarantees more stability, it leads to conservative actions, limiting the impact of the agent's decision. Therefore, we can consider other heuristics, such as oscillation reduction, and augment domain-specific information, e.g., whether the cluster has capacity to fulfill the requests. Furthermore, we can enable more purposeful actions by gathering data over a more extended timeframe (e.g., 30 minutes), by performing loops and collecting reward locally, and only actually update the thresholds after the action is consolidated. Another limitation is that the success of the scaling mechanism, being it static or relying on RL, depends on the initial threshold value. For this reason, in the next steps we plan to integrate the autoscaler with profiling information [19, 20] that can help to have a good understanding of what the general runtime patterns are. Furthermore, leveraging runtime predictions [13, 21] on the SLO fulfillment can aid the RL agent to take more informed decisions. In this direction, we could explore expanding our HPAQT to be model-based, i.e., include a wider understanding of the environment, by leveraging side modeling. This approach can allow to have more control over the domain, with the side effect of introducing higher computing complexity. Finally, while the current evaluation setup offers a compelling test scenario, extending the analysis to more applications and infrastructures will offer a more robust assessment of HPAQT.

6 Conclusion

This paper presents HPAQT, an innovative approach to autoscaling. HPAQT is motivated by the need for trustworthy, low-overhead autoscaling that self-calibrates thresholds to meet high-level SLOs—improving compliance and cost without sacrificing interpretability. One of our core contributions lies in developing a reinforcement learning (RL) augmentation for autoscaling, allowing for a more flexible target definition, guaranteeing to map high-level requirements to appropriate actions at the resource level,

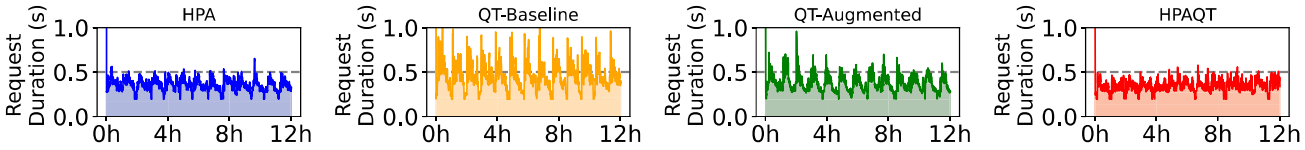


Figure 6: Request duration values for the different algorithms over the observation period.

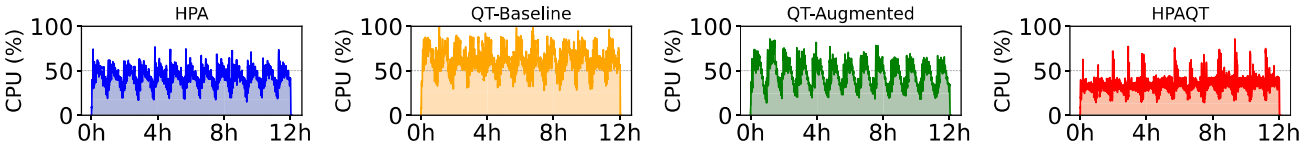


Figure 7: Comparative plots of CPU usage in the observation period.

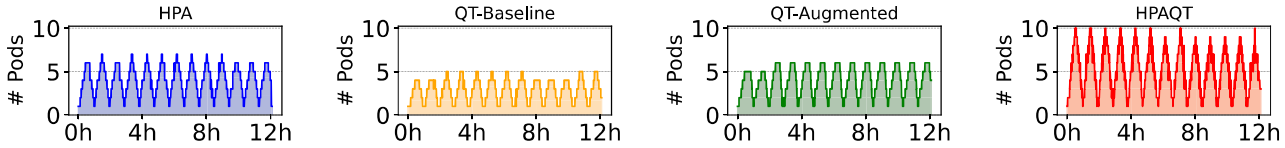


Figure 8: Comparative analysis of autoscaling performance, as the number of active pods over time.

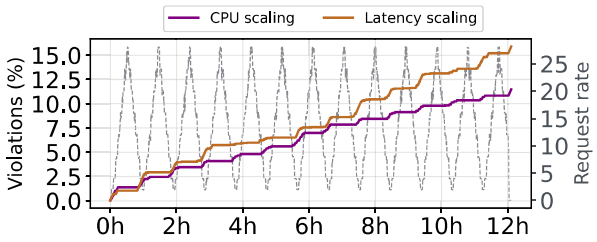


Figure 9: HPAQT violations on the target cost efficiency metric, using CPU or latency as a scaling metric.

and improving the adaptability and reusability of the autoscaler. The framework's ability to define and use high-level intents significantly enhances the autoscaler's functionality. Centrally, we mitigated the issues in defining the RL agent's state, actions, and rewards by simplifying the RL agent's control over the threshold and employing a lightweight Q-learning approach. Additionally, the design contemplates the complexities of building an autoscaler independent of Horizontal Pod Autoscaler (HPA) norms, incorporating strategies to reduce oscillation and improve scaling decisions. A key aspect of our approach is the emphasis on interpretability in autoscaling. By adopting a simplified approach and maintaining a small state/action space, our research advances the interpretability of RL-based autoscaling. This simplicity, derived from industry-standard approaches, ensures ease of understanding and fosters trust in deploying such autoscalers in practical applications. Future research will focus on advancing Q-Threshold approaches by investigating alternative RL algorithms to refine threshold control. Another promising area of exploration is the decoupling of the autoscaling control loop. This could involve extending data collection timeframes and updating thresholds less frequently to optimize efficiency and accuracy.

Acknowledgments

This work is funded by the HORIZON Research and Innovation Action 101135576 INTEND "Intent-based data operation in the computing continuum."

References

[1] Siddharth Agarwal, Maria A. Rodriguez, and Rajkumar Buyya. 2024. A Deep Recurrent-Reinforcement Learning Method for Intelligent AutoScaling of Serverless Functions. *IEEE Transactions on Services Computing* 17, 5 (Sept. 2024), 1899–1910. doi:10.1109/TSC.2024.3387661

[2] Hamid Arabnejad, Claus Pahl, Pooyan Jamshidi, and Giovani Estrada. 2017. A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 64–73. doi:10.1109/CCGRID.2017.15

[3] Bartosz Balis, Tomasz Bartynski, Marian Bubak, Daniel Harezlak, Marek Kasztelnik, Maciej Malawski, Piotr Nowakowski, Maciej Pawlik, and Bartosz Wilk. 2017. Smart levee monitoring and flood decision support system: reference architecture and urgent computing management. *Procedia computer science* 108 (2017), 2220–2229.

[4] Priscilla Benedetti, Mauro Femminella, Gianluca Reali, and Kris Steenhaut. 2022. Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications. In *2022 IEEE international conference on pervasive computing and communications workshops and other affiliated events (PerCom Workshops)*. IEEE, 674–679.

[5] Miltiadis Chrysopoulos, Ioannis Konstantinou, and Nectarios Koziris. 2023. Deep Reinforcement Learning in Cloud Elasticity Through Offline Learning and Return Based Scaling. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, Chicago, IL, USA, 13–23. doi:10.1109/CLOUD60044.2023.00012

[6] Patrizio Dazzi, Luca Ferrucci, Marco Danelutto, Konstantinos Tserpes, Antonios Makris, Theodoros Theodoropoulos, Jacopo Massa, Emanuele Carlini, and Matteo Mordacchini. 2024. Urgent edge computing. In *Proceedings of the 4th Workshop on Flexible Resource and Application Management on the Edge*. 7–14.

[7] Nikos Filinis, Ioannis Tzanettis, Dimitrios Spatharakis, Eleni Fotopoulou, Ioannis Dimolitsas, Anastasios Zafeiropoulos, Constantinos Vassilakis, and Symeon Papavassiliou. 2024. Intent-driven orchestration of serverless applications in the computing continuum. *Future Generation Computer Systems* 154 (May 2024), 72–86. doi:10.1016/j.future.2023.12.032

[8] Manuel Gotin, Felix Lösch, Robert Heinrich, and Ralf Reussner. 2018. Investigating performance metrics for scaling microservices in cloudiot-environments. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. 157–167.

[9] Tor Atle Hjeltnes and Borje Hansson. 2005. Cost effectiveness and cost efficiency in e-learning. *QUIS-Quality, Interoperability and Standards in e-learning, Norway* 34 (2005).

[10] Shay Horovitz and Yair Arian. 2018. Efficient Cloud Auto-Scaling with SLA Objective Using Q-Learning. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*. 85–92. doi:10.1109/FiCloud.2018.00020

[11] Abeer Abdel Khaleq and Ilkyeun Ra. 2021. Intelligent autoscaling of microservices in the cloud for real-time applications. *IEEE access* 9 (2021), 35464–35476.

[12] Ezgi Korkmaz. 2024. A survey analyzing generalization in deep reinforcement learning. *arXiv preprint arXiv:2401.02349* (2024).

[13] Anna Lackinger, Andrea Morichetta, and Schahram Dustdar. 2024. Time series predictions for cloud workloads: A comprehensive evaluation. In *2024 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 36–45.

[14] Allan Lago, Sahaj Patel, and Aditya Singh. 2024. Low-cost real-time aerial object detection and GPS location tracking pipeline. *ISPRS Open Journal of Photogrammetry and Remote Sensing* 13 (2024), 100069.

[15] Zheng Li, Liam O'brien, He Zhang, and Rainbow Cai. 2012. On a catalogue of metrics for evaluating commercial cloud services. In *2012 ACM/IEEE 13th International Conference on Grid Computing*. IEEE, 164–173.

[16] Hongyun Liu, Peng Chen, and Zhiming Zhao. 2021. Towards A Robust Meta-Reinforcement Learning-Based Scheduling Framework for Time Critical Tasks in Cloud Environments. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, Chicago, IL, USA, 637–647. doi:10.1109/CLOUD53861.2021.00082

[17] Thijs Metsch, Magdalena Viktorsson, Adrian Hoban, Monica Vitali, Ravi Iyer, and Erik Elmroth. 2023. Intent-driven orchestration: Enforcing service level objectives for cloud native deployments. *SN Computer Science* 4, 3 (2023), 268.

[18] Pratik Mishra, Sandeep Hans, Diptikalyan Saha, and Pratibha Moogi. 2024. Optimizing Cloud Workloads: Autoscaling with Reinforcement Learning. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*. IEEE, Shenzhen, China, 217–222. doi:10.1109/CLOUD62652.2024.00033

[19] Andrea Morichetta, Victor Casamayor-Pujol, Stefan Nastic, Schahram Dustdar, Deepak Vij, Ying Xiong, and Zhaobo Zhang. 2023. PolarisProfiler: A Novel Metadata-Based Profiling Approach for Optimizing Resource Management in the Edge-Cloud Continnum.. In *SOSE*. 27–36.

[20] Andrea Morichetta, Stefan Nastic, Victor Casamayor Pujol, and Schahram Dustdar. 2025. Formal and Empirical Study of Metadata-Based Profiling for Resource Management in the Computing Continuum. *arXiv preprint arXiv:2504.20740. To appear in ACM Transactions On Internet Technology (minor revision)*. (2025).

[21] Andrea Morichetta, Thomas Pusztai, Deepak Vij, Víctor Casamayor Pujol, Philipp Raith, Ying Xiong, Stefan Nastic, Schahram Dustdar, and Zhaobo Zhang. 2023. Demystifying deep learning in predictive monitoring for cloud-native SLOs. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 1–11.

[22] Seyed Mohammad Reza Nouri, Han Li, Srikumar Venugopal, Wenxia Guo, MingYun He, and Wenhong Tian. 2019. Autonomic Decentralized Elasticity Based on a Reinforcement Learning Controller for Cloud Applications. *Future Generation Computer Systems* 94 (May 2019), 765–780. doi:10.1016/j.future.2018.11.049

[23] Thomas Pusztai, Andrea Morichetta, Víctor Casamayor Pujol, Schahram Dustdar, Stefan Nastic, Xiaoning Ding, Deepak Vij, and Ying Xiong. 2021. A Novel Middleware for Efficiently Implementing Complex Cloud-Native SLOs. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 410–420. doi:10.1109/CLOUD53861.2021.00055

[24] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 805–825.

[25] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. 2023. {AWARE}: Automate workload autoscaling with reinforcement learning in production cloud systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 387–402.

[26] Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti, and Matteo Nardelli. 2022. Dynamic multi-metric thresholds for scaling applications using reinforcement learning. *IEEE Transactions on Cloud Computing* 11, 2 (2022), 1807–1821.

[27] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. 2019. Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 329–338. doi:10.1109/CLOUD.2019.00061

[28] Lucia Schuler, Somaya Jamil, and Niklas Kühl. 2020. AI-based Resource Allocation: Reinforcement Learning for Adaptive Auto-scaling in Serverless Environments. doi:10.48550/arXiv.2005.14410 arXiv:2005.14410 [cs].

[29] Geng Sun, Long He, Zemin Sun, Qingqing Wu, Shuang Liang, Jiahui Li, Dusit Niyato, and Victor CM Leung. 2024. Joint task offloading and resource allocation in aerial-terrestrial UAV networks with edge and fog computing for post-disaster rescue. *IEEE Transactions on Mobile Computing* 23, 9 (2024), 8582–8600.

[30] Pengcheng Tang, Fei Li, Wei Zhou, Weihua Hu, and Li Yang. 2015. Efficient Auto-Scaling Approach in the Telco Cloud Using Self-Learning Algorithm. In *2015 IEEE Global Communications Conference (GLOBECOM)*. 1–6. doi:10.1109/GLOCOM.2015.7417181

[31] Yi Wei, Daniel Kudenko, Shijun Liu, Li Pan, Lei Wu, and Xiangxu Meng. 2019. A Reinforcement Learning Based Auto-Scaling Approach for SaaS Providers in Dynamic Cloud Environment. *Mathematical Problems in Engineering* 2019 (Feb. 2019), e5080647. doi:10.1155/2019/5080647

[32] Siqiao Xue, Chao Qu, Xiaoming Shi, Cong Liao, Shiyi Zhu, Xiaoyu Tan, Lintao Ma, Shiyu Wang, Shijun Wang, Yun Hu, Lei Lei, Yangfei Zheng, Jianguo Li, and James Zhang. 2022. A Meta Reinforcement Learning Approach for Predictive Autoscaling in the Cloud. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM, Washington DC USA, 4290–4299. doi:10.1145/3534678.3539063

[33] Zhiyu Zhang, Tao Wang, An Li, and Wenbo Zhang. 2022. Adaptive Auto-Scaling of Delay-Sensitive Serverless Services with Reinforcement Learning. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, Los Alamitos, CA, USA, 866–871. doi:10.1109/COMPSAC54236.2022.00137