AWARE: Automate Workload Autoscaling with Reinforcement Learning in Production Cloud Systems

Haoran Qiu[1] Weichao Mao[1] Chen Wang[2] Hubertus Franke[2] Alaa Youssef[2]
Zbigniew T. Kalbarczyk[1] Tamer Başar[1] Ravishankar K. Iyer[1]

[1]*University of Illinois at Urbana-Champaign* [2]*IBM Research*

Abstract

Workload autoscaling is widely used in public and private cloud systems to maintain stable service performance and save resources. However, it remains challenging to set the optimal resource limits and dynamically scale each workload at runtime. Reinforcement learning (RL) has recently been proposed and applied in various systems tasks, including resource management. In this paper, we first characterize the state-of-the-art RL approaches for workload autoscaling in a public cloud and point out that there is still a large gap in taking the RL advances to production systems. We then propose AWARE, an extensible framework for deploying and managing RL-based agents in production systems. AWARE leverages meta-learning and bootstrapping to (a) automatically and quickly adapt to different workloads, and (b) provide safe and robust RL exploration. AWARE provides a common OpenAI Gym-like RL interface to agent developers for easy integration with different systems tasks. We illustrate the use of AWARE in the case of workload autoscaling. Our experiments show that AWARE adapts a learned autoscaling policy to new workloads 5.5× faster than the existing transfer-learning-based approach and provides stable online policy-serving performance with less than 3.6% reward degradation. With bootstrapping, AWARE helps achieve 47.5% and 39.2% higher CPU and memory utilization while reducing SLO violations by a factor of 16.9× during policy training.

1 Introduction

Motivation. Reinforcement learning (RL) has become an active area in machine learning research and is widely used in various systems tasks (e.g., resource scaling [23,47–49,59,62], power management [58, 64], job scheduling [4, 5, 32, 33, 35, 63], video streaming [34, 60], and congestion control [22, 28, 31, 56, 60]). As a viable alternative to human-generated heuristics, RL automates the repetitive process of heuristics tuning and testing by enabling an *agent* to learn the optimal *policy* directly from interaction with the *environment*.

One example is workload resource autoscaling for meeting application service-level objectives (SLOs) while achieving high resource utilization efficiency [8, 30, 47, 49, 50, 59]. Traditional rule-based approaches [2, 3, 6, 25] configure static upper and lower thresholds offline for certain system metrics (e.g., CPU or memory utilization) or application metrics (e.g., request arrival rate, throughput, or end-to-end latency) so that resources can be scaled accordingly when the measured metrics go above or below the thresholds. Tuning and testing of fine-grained thresholds require significant application/system-specific domain knowledge from experts to achieve optimal resource allocation. Further, repeated parameter tuning for each workload can be labor-intensive, especially for microservice-like applications in large-scale production systems. As different types of services may use different amounts of resources (e.g., CPU and memory) and are sensitive to different kinds of interference and workload spikes, a customized threshold has to be set for a service differently.

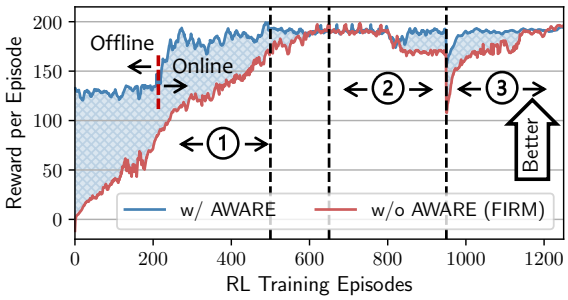

Figure 1: RL agent performance when managed by AWARE compared to the baseline (FIRM [47]). Stages ①, ②, and ③ demonstrate the benefit of RL bootstrapping, incremental retraining, and fast adaptation, respectively.

RL, on the other hand, is well-suited for learning optimal policies, as it models a systems task (e.g., workload autoscaling) as a sequential decision-making problem and provides a tight feedback loop for exploring the state-action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules) [32, 48]. Integrating RL with those complex systems management tasks in production systems can (a) make full use of the abundant monitoring data on applications and the infrastructure, and (b) automate the process of developing optimal policies while freeing operators from manual workload profiling and parameter tuning/testing. For example, FIRM [47]'s RL agent learns an optimal workload autoscaling policy that adapts to specific application workloads with online telemetry data that alleviates the need for handcrafted heuristics (see §2.2 for details).

Challenges. However, even as RL is starting to show its strength in the systems and networking domains [4, 5, 22, 23, 28, 31–35, 47–49, 56, 58–60, 62–64], there is still a large

gap in directly applying RL advances to real-world production systems due to a series of assumptions that are rarely satisfied in practice. First, a learned RL policy is workload-specific and infrastructure-specific. Retraining is needed to adapt to a new workload or underlying infrastructure in heterogeneous and dynamically evolving (possibly multi-cloud) datacenters [18,37,53,54,58]. For instance: (a) In SLO-driven resource management, application performance and utilization differ significantly among heterogeneous workloads [47]. Fig. 1 stage ③ shows that FIRM's trained RL policy suffers performance degradation and requires substantial retraining. (b) In power management, diverse power consumption and workload sensitivity to core/uncore frequency require separate training of RL policies [58]. (c) In video streaming and network congestion control, different sets of traces have diverse payload characteristics and network environments [60] (e.g., dynamic link bandwidth, delay, and loss rate). Even with transfer learning (TL) [47], nontrivial retraining is needed to adapt to new workloads and environment shifts in each problem domain, which is a critical problem in making RL practical in production. Further, TL requires fine-grained environment clustering to identify the most appropriate model to transfer from, and requires saving one model per cluster.

Second, for the same application and environment, there could be slight changes (e.g., patches and rolling updates), unusual workload patterns not seen before (e.g., due to migration rollout), or traffic jitters. Without timely retraining, the online policy-serving performance of the RL agent fluctuates and leads to undesired degradation (as shown in Fig. 1 Stage ②). It is crucial to ensure robust online performance in case of environment or model uncertainty [40,46].

Third, RL training is through trial and error [32,35,47], so worse-than-baseline or suboptimal decisions can be generated, especially at an early stage of training (as shown in Fig. 1 Stage ①). Direct training in the production system leads to suboptimal performance and undesired SLO violations, while training in a simulator and then transitioning to the production system face the problem of poor generalization [61].

A framework that can bring the RL advances to production systems is needed so that (a) the RL model can be trained in a safe and robust manner, (b) the learned RL policy can be adapted to new workloads and altered environments seamlessly without significant retraining, and (c) the online RL model policy-serving performance can be kept stable.

Our Work. We first performed a characterization study of RL in production systems in the task of workload autoscaling. The study focused on the impact of workload change and environment shift regarding (a) RL agent performance degradation or variation and (b) retraining cost. To facilitate the deployment and operation of RL agents in the systems management tasks of a production cloud environment, we introduce an RL model-serving and management framework. As a general framework to support a variety of RL agents for systems management tasks, it can be used by system operators to develop RL-based agents that can be quickly adapted to new environments and achieve stable online policy-serving performance with continuous monitoring and safe bootstrapping (as shown in Fig. 1). In the end, system operators can benefit from RL automation of systems management tasks.

- To achieve **fast model adaptation** in each domain or systems task, we leverage meta-learning [39] to model the RL agent as a *base-learner* and create a *meta-learner* for learning to generalize and adapt to new applications and environment shifts. The base-learner discovers policies that generalize across workload variations and intra-environment dynamicity for an <application, environment> pair, while the meta-learner generalizes across <application, environment> pairs to address inter-environment dynamicity and application heterogeneity. We designed a novel framework that allows the meta-learner to learn to generate an *embedding* [38, 39, 57] that projects the application- and system-specific data to a vector space. On this projected vector space, workloads with similar characteristics are projected to closer locations, while those with quite different characteristics are projected to locations far from each other. The embedding is generated by encoding a set of *episodes* from the RL agent's exploration of the environment. Since each episode records a step-by-step interaction of the RL agent with the environment, the time-series episodes naturally encode spatial and temporal characteristics. In the task of workload autoscaling, spatial characteristics correspond to the workload's performance sensitivity to different resource allocations, and temporal characteristics correspond to the time-varying load patterns. The generated embedding is then fed as input to the base-learner to adapt to the application and environment shift (from the environments with similar characteristics). With the embedding, fewer retraining iterations are needed for new, previously unseen workloads.
- To achieve **stable online RL policy-serving performance**, we leverage continuous monitoring, and designed a retraining detection and trigger mechanism. An RL agent observes a *state*, performs an *action*, and gets a *reward* at every step in an episode. The time series of states, actions, and rewards in an episode form a *trajectory*. RL trajectories are collected and stored in a time-series database. The most recent rewards are used to calculate the average reward and variation for comparison against user-specified targets. Continuous monitoring ensures that RL model retraining can be triggered or stopped timely so that the RL policy can seamlessly adapt to any environment jitters. We intercept the RL model update logic to enable the switch between RL policy serving and retraining.
- To achieve **safe RL exploration**, we designed an RL bootstrapping module that combines offline and online training. The agent starts with offline training, and a traditional heuristics-based controller (e.g., the Kubernetes Horizontal Pod Autoscaler (HPA) [25] and the Vertical Pod Autoscaler (VPA) [15] in the case of workload autoscaling), is used as the navigator for (online) exploration of the state and action space in the environment. After the RL model is trained to the

same level as the heuristics-based controller by comparing rewards, the agent continues to be trained online.

We have demonstrated the proposed framework in the task of workload autoscaling on Kubernetes by implementing AWARE (i.e., **A**utomate **W**orkload **A**utoscaling with **RE**inforcement learning). Each RL agent manages a Kubernetes Deployment and configures resources automatically, adjusting both the number of replicas (horizontal scaling) and the CPU/memory limits (vertical scaling) to maintain workload service-level objectives (SLOs) and achieve high resource utilization. To integrate RL agents with Kubernetes, we designed and implemented a multidimensional Pod autoscaler (MPA) system. MPA provides system support for RL-based controllers and translates RL outputs into multidimensional autoscaling actions in a holistic manner by (a) providing an API for RL agents to execute horizontal and vertical scaling decisions on Pod CPU and memory limits, (b) combining vertical and horizontal scaling actions in a single CRD object [24], and (c) providing a user interface for user-defined objective functions for multidimensional autoscaling.

Results. We present a detailed experimental evaluation of AWARE, demonstrating that AWARE significantly improves the practicality of applying RL in production cloud systems (for workload autoscaling). We first show that the adaptation process of a learned autoscaling policy to new workloads with meta-learning is 5.5× faster than the existing transfer-learning-based approach (§5.2), and then demonstrate that AWARE provides stable online policy-serving performance with less than 3.6% reward degradation (§5.3). AWARE's bootstrapping mechanism helps achieve 47.5% and 39.2% higher CPU and memory utilization while reducing SLO violations by a factor of 16.9× during training (§5.4).

Contributions. In summary, our main contributions are:

- A characterization of RL-based production workload autoscaling and the challenges involved in applying RL in production cloud systems (§2.3).
- The design of a novel meta-learning-based framework for fast RL model adaptation in workload autoscaling (§3.2).
- The design of an RL retraining management and bootstrapping mechanism for stable policy-serving performance (§3.3) and robust RL environment exploration (§3.4).
- An implementation of the proposed framework in the task of workload autoscaling with MPA, which enables integration of RL agents with Kubernetes (§4.1).
- A detailed evaluation of AWARE that demonstrates substantial improvements through meta-learning and RL lifecycle management while maintaining workload SLOs and resource utilization (§5).

2 Background & Characterization

2.1 Reinforcement Learning

In reinforcement learning (RL), an *agent* interacts with an *environment* modeled as a discrete-time Markov decision process (MDP) (as shown in Fig. 2). At time step t, the agent

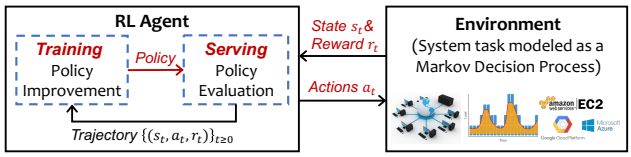

Figure 2: An RL agent interacting with an environment modeled as a systems task (e.g., workload autoscaling or congestion control) in the form of a Markov decision process (MDP).

perceives a *state* $s_t \in S$ of the environment and takes an *action* $a_t \in A$. The agent receives a *reward* $r_t \in \mathbb{R}$ as feedback on how good the decision is, and at the next time step $t+1$, the environment transitions to a new state s_{t+1}. The whole sequence of transitions $\{(s_t, a_t, r_t)\}_{0\leq t\leq T}$ is called a *trajectory* or *episode* of length T. The agent's goal is to learn a *policy* π_θ[1] that maximizes the expected cumulative rewards in the future, i.e., $\mathbb{E}[\sum_{t=0}^{T} \gamma^t \cdot r_t]$, where the discount factor $\gamma \in (0,1)$ progressively de-emphasizes future rewards. RL consists of a *policy-training* stage and a *policy-serving* stage [41]. At the policy-training stage, the agent (using an initialized policy) starts with no knowledge about the task and learns by reinforcement and directly interacting with the environment. At the policy-serving stage, the trained policy is used to generate an action based on the current state of the environment, and model parameters are no longer being updated.

2.2 Workload Autoscaling with RL

Because of the sequential nature of the decision-making process, RL is well-suited for learning resource management policies, as it provides a tight feedback loop for exploring the state-action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules) [32, 48]. In addition, since the decisions made for workloads are highly repetitive, an abundance of data is generated to train such RL algorithms even with deep neural networks[2]. By directly learning from the actual workload and operating conditions to understand how the allocation of resources affects application performance, the RL agent can optimize for a specific workload and adapt to varying conditions in the learning environment. RL [23, 47–49, 59, 62] has been shown to automate resource management and outperform heuristics-based approaches in terms of meeting workload SLOs and achieving higher resource utilization.

Specifically, we adopted the design and took the open-source implementation of an RL-based workload autoscaler from FIRM [47], which is the state-of-the-art RL-based autoscaling solution, to the best of our knowledge. FIRM uses an actor-critic RL algorithm called DDPG [29].

The RL agent monitors the system- and application-specific measurements and learns how to scale the allocated resources vertically and horizontally. Table 1 shows the model's state

[1] A policy π_θ maps the state space S to the action space A and is usually represented by neural networks (with parameters denoted by θ).

[2] Deep neural networks can express complex system-application environment dynamics and decision-making policies but are data-hungry.

Table 1: State-action space of the RL agent.

State Space (s_t)
Resource Limits (CPU, RAM), Resource Utilization (CPU, Memory, I/O, Network), SLO Preservation Ratio (Latency, Throughput), Observed Load Changes
Action Space (a_t)
Resource Limits (CPU, RAM), Number of Replicas

Table 2: RL training hyperparameters.

Parameter	**Value**
# Time Steps per Episode	100×64 mini-batches
Replay Buffer Size	10^6
Learning Rate	Actor (3×10^{-4}), Critic (3×10^{-3})
Discount Factor	0.99
Soft Update Coefficient	3×10^{-3}
Random Noise	μ (0), σ (0.2)
Exploration Factor	ε (1.0), ε-decay (10^{-6})

and action spaces. The goal is to achieve high resource utilization (RU) while maintaining application SLOs (if there are any). SLO preservation (SP) is defined as the ratio between the SLO metric and the measured metric. If no SLO is defined for the workload (e.g., best-effort jobs) or the measured metric is smaller than the SLO metric, $SP = 1$. An SLO metric can be either request serving latency (e.g., the 99th percentile of the requests are completed in 100ms) or throughput (e.g., request processing rate is no less than 100/s). The reward function is then defined the same as in FIRM [47], $r_t = \alpha \cdot SP_t \cdot |\mathcal{R}| + (1-\alpha) \cdot \sum_{i \in \mathcal{R}} RU_i$, where $\mathcal{R}$ is the set of resources. Table 2 lists the hyperparameters tuned for better performance in the experiments. The RL algorithm is trained in an episodic setting. In each episode, the agent manages the autoscaling of the application workload for a fixed period of time (100 RL time steps in our experiments).

2.3 Characterization of RL in Production

In the characterization study of FIRM for workload autoscaling, we selected 16 representative production cloud workloads based on a survey of 89 industry use cases of serverless computing applications [11], as serverless workloads are highly dynamic (and thus require autoscaling) and rely on the provider to manage the resources. The selected production workloads include CPU-intensive tasks (e.g., floating-point number computation), image manipulation, text processing, data compression, web serving, ML model serving, and I/O services (e.g., read, write, and streaming). Next, we deployed the selected workloads as Deployments in a five-node Kubernetes cluster in a public cloud and ran an RL-based multi-dimensional autoscaler (i.e., a FIRM agent) with each workload. All nodes run Ubuntu 18.04 with four cores, 16 GB memory, and a 200 GB disk. For RL agent training and inference, we used real-world datacenter traces [65] released by Microsoft Azure, collected over two weeks in 2021.

We next present the key insights from the characterization study in the order of adaptation, online policy-serving, and early-stage of RL training.

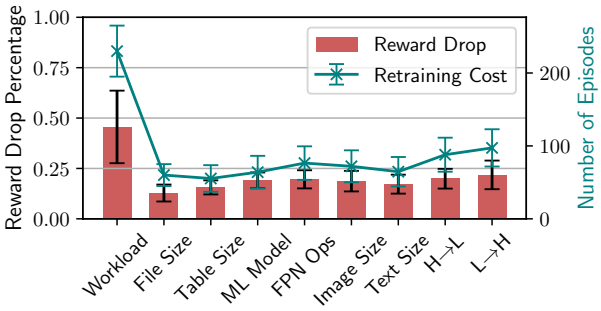

Figure 3: Retraining cost of RL models.

Insight 1: Adaptation Retraining Cost. To study the retraining cost of adapting a trained RL policy to new application workloads, we selected each application from the workload pool, trained an RL agent for the application until convergence, and retrained the learned RL policy to serve all the other different applications. We then measured the reward drop after the workload changed and the number of episodes each agent took to retrain to convergence. As shown in Fig. 3 (column 1), we observed a 45.6% average per-episode reward drop percentage when the workload had been changed, and retraining to convergence required around 230 episodes (with the model parameter transfer learning used in FIRM [47]).

Insight 2: Online Policy-serving Performance Jitters. We introduced seven scenarios to explore the performance instability of RL-based workload autoscaling agents when facing application or service payload size changes and load pattern changes. For I/O services to a backend file system (e.g., AWS S3) and the compression/decompression services, the size of files being read, written, or streaming was changed from [128 KB, 256 KB, 384 KB] to [512 KB, 768 KB, 1024 KB]. For database services, the size of the table being scanned was changed from 1024 items to 10240 items. For floating-point number calculation, the number of operations was changed from 10^8 to 20^8. For image manipulations, the dimension was changed from 40×40 to 160×160. For text processing, the JSON file size was changed from [250 B, 500 B, 1 KB] to [2 KB, 3 KB, 5 KB]. For ML model serving, we changed the matrix multiplication dimension from 50 to 150. For load pattern changes, we divided the Azure workload traces into two parts, one half with a higher daily load ($> 10^5$ per day) and the other half with a lower daily load ($\leq 10^5$ per day).

Fig. 3 (columns 2–9) shows the per-episode reward drop percentage and the retraining cost of each scenario. File size changes led to the lowest 12.8% reward drop and around 70 episodes of retraining. We attribute this to I/O-intensive workloads' relatively low sensitivity to CPU/memory allocation, compared to compute- or memory-intensive workloads. Other payload-related changes (i.e., table size, ML model, floating-point number operations, image dimension, and text size) resulted in a 15.6–19.9% reward drop. Load changes from high request arrival rates to low arrival rates (i.e., H→L

Table 3: Workload performance and utilization efficiency deficit (i.e., the relative difference compared to the rule-based approach) in early-stage RL model training.

RL Episodes	EP 1–100	EP 101–200	EP 201–300	EP 301–400
CPU Util	-32.3% ± 14%	-42.9% ± 15%	-22.1% ± 12%	-10.0% ± 6%
Memory Util	-28.8% ± 11%	-30.5% ± 10%	-26.5% ± 8%	-7.8% ± 2%
SLO Violations	56.1 ± 14 ×	22.2 ± 7 ×	12.7 ± 5 ×	10.1 ± 3 ×

in Fig. 3) and from low rates to high rates (i.e., L→H) resulted in 19.9% and 21.8% reward drops, which required around 98 and 107 episodes of model retraining, respectively.

Insight 3: Cost of Early-stage RL Training. As mentioned in §2.2, RL training proceeds in episodes. When the initialized RL agent starts to learn the optimal policy, especially at an early stage of policy training, the policy might be worse than the baseline heuristics-based approach or even produce undesired actions, such as an oscillating scaling up and down behavior. This is primarily due to the exploration of the state-action space and RL agent learning through trial and error. To study what is lost during policy training, we compared workloads managed by RL agents with the same workloads managed by the rule-based autoscaling approach (i.e., HPA and VPA). We define the early stage of RL training to be the training process from the beginning to the episode at which the RL agent starts to get better than the rule-based approach (which is around 400 episodes in our experiments) because we are interested in the loss due to RL training compared to non-RL-based approaches. We then divided the 400 episodes (in the early training stage) into four segments. For each segment, we calculated the accumulated utilization deficit and SLO violations of the application workloads controlled by the RL agents; the results are shown in Table 3. The relative difference in utilization or SLO performance is based on the comparison between the RL agent and the rule-based approach when used to control the same application workloads with the same set of traces.

Results show that RL policies necessarily lead to poor decisions in the early stages of training. In the first 100 episodes, the RL agents inevitably caused more SLO violations than in the other segments (56.1× more than the rule-based approach, which had five SLO violations per 100 episodes). We observe that most SLO violations were due to the under-provisioning of resources, so the CPU and memory utilization deficits (32.3% and 28.8% lower, respectively, than for the rule-based approach) were smaller than those in the later segments. In the last three segments, we observe a utilization deficit (i.e., 10–42.9% lower CPU utilization and 7.8–30.5% lower memory utilization) and more SLO violations (i.e., 10.1–22.7×) compared to the rule-based approach.

Summary and Implications. Workloads running in production cloud systems might be user-facing or high-stakes. *To enjoy the benefit of RL in systems management, the key challenge is to produce fast-adapting, effective, and robust RL-based solutions under the constraints of production cloud systems.* As of now, to the best of our knowledge, there are no systems that can help agent developers address this challenge.

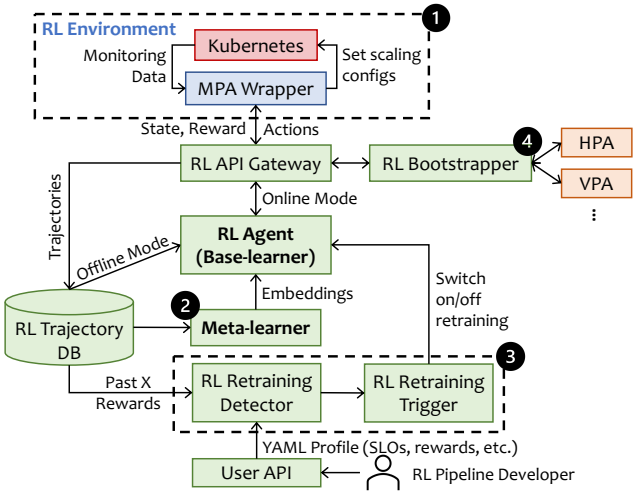

Figure 4: Overview of AWARE.

3 AWARE Design

3.1 Overview

Driven by the insights from §2.3, we describe the design of AWARE, a framework that supports RL agents for multidimensional Pod autoscaling (MPA) of workloads in production Kubernetes systems. AWARE manages the RL agent lifecycle to deliver stable and robust agent performance. Fig. 4 provides an overview of AWARE. We next present a brief summary of each component in this section.

RL Environment. The RL environment (denoted by ❶ in Fig. 4) of AWARE consists of a cluster deployment (e.g., Kubernetes) and an MPA wrapper. The MPA wrapper is designed and implemented as a shim layer that follows an "agent-centric" pattern of request-response interaction advocated by OpenAI Gym [44]. The purpose of the MPA wrapper is to translate measurements and scaling recommendations to and from RL abstractions (i.e., states/rewards and actions), respectively. The communication between the wrapper and the RL agent is through remote procedure calls (RPCs). When the agent steps the environment forward by sending an action to the MPA wrapper through the RPC request, the wrapper translates the received action to vertical and horizontal scaling configurations and applies it to the cluster deployments (e.g., by setting the VPA object [15] and calling the replica re-scaling API). The wrapper gets measurements from the monitoring service in the cluster (e.g., Prometheus [7] in Kubernetes), translates them to RL states and rewards, and sends them back to the agent through the RPC response. The wrapper then waits on the RPC server for the next action request. We describe implementation details in §4.1.

The framework can also be applied to other systems management tasks (e.g., job scheduling or network congestion control) by replacing the RL environment. Decoupling the RL environment (i.e., the environment wrapper) from the rest of the framework and using the standard OpenAI Gym interface

make environment replacement easy [33].

RL API Gateway. The RL API gateway connects the RL agent to the MPA wrapper by sending the RL action in an RPC request and unpacking the state and reward in the RPC response for the RL agent. Each RL trajectory consists of <state, action, reward> transitions in one episode where the RL environment defines the length or the terminating condition of an episode. The trajectories from each RL agent, along with the logical timestamp (i.e., the episode and time step index), are saved to the RL trajectory database.

RL Agent (Base-learner). The RL agent implements the DDPG RL algorithm (as described in FIRM [47]) and interacts with the RL environment to perform policy training or policy serving (i.e., inference). Since the interface between the RL agent and the MPA wrapper follows the OpenAI Gym standard, different advanced RL algorithms can be used to replace the original RL algorithm DDPG.

Meta-learner. To help adapt to new workloads or environment updates within the problem domain, the meta-learner (denoted by ❷) selects RL trajectories from the database and generates an embedding that accurately represents the workload running in the environment. RL trajectories are selected per application, and the criteria are based on the reward associated with each trajectory. The embedding is then fed to the base-learner (i.e., the RL agent) as part of the input. The RL agent leverages the embedding to adapt (fine-tune) its policy by differentiating heterogeneous workloads and environment updates. See §3.2 for more details.

RL Retraining Detector and Trigger. At the end of each episode, the RL retraining detector (denoted by ❸) pulls the recent episode rewards gained by the agent from the trajectory database. The mean and standard deviation of the per-episode rewards are calculated and compared to predefined thresholds for performance and variability assessment. If conditions are met, the RL retraining trigger will intercept the inference or training loop of the RL agent to switch retraining on or off, respectively. See §3.3 for more details.

RL Bootstrapper. The RL bootstrapper (denoted by ❹) determines whether the RL training is online or offline. In the online RL training mode, the RL agent interacts directly with the RL environment. However, offline RL training avoids worse-than-baseline performance or illegal actions in the early stages of RL training, which is desired by production systems. In the offline RL training mode, the RL policy training happens offline based on data collected using a fallback option (i.e., a heuristics-based method), while the RL policy is not used for interacting with the environment. The RL bootstrapper intercepts the request-response path between the RL agent and the RL API gateway and replaces the RL agent with the controller implemented as the fallback option. For instance, in the case of workload autoscaling, the default autoscalers widely used are the traditional rule-based approaches HPA (for horizontal scaling) and VPA (for vertical scaling). Given the states at each time step, corresponding autoscaling actions

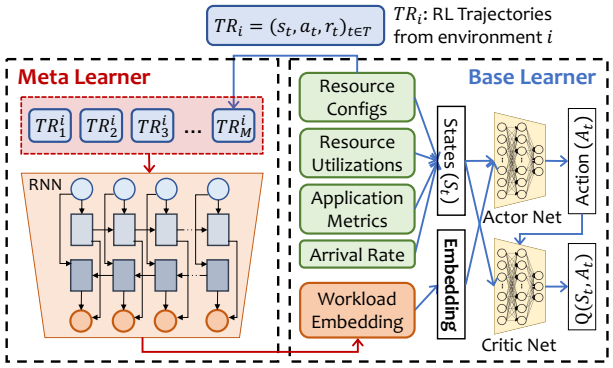

Figure 5: Architecture of meta-learning for RL.

are then generated based on the HPA and VPA algorithms and sent back to the RL API gateway for execution. The trajectories recorded in the RL trajectory database will be used for the offline RL policy training. See §3.4 for more details.

3.2 Meta-learner

Traditional RL-based resource management approaches [23, 47, 48, 59, 62] require the collection of large amounts of training data samples and retraining (even with transfer learning) to adapt to new environments for (a) updated or previously unseen application workloads or (b) constantly evolving cloud infrastructures [18, 37, 53, 58]. Pure RL-based approaches are no longer tenable in such dynamic cloud environments or even in the context of multi-cloud computing [54]. A novel approach that provides fast model adaptation is needed to make RL practical in production cloud systems.

In AWARE, we leverage meta-learning to reduce the retraining overhead and thus adapt quickly to new environments. In essence, the RL agent is treated as the *base-learner* for an individual environment, and a *meta-learner* is designed to generate representative *embeddings* that help differentiate environments. We next give a brief primer on meta-learning and the concept of embeddings before presenting AWARE's meta-learning model and an interpretation of embeddings from a systems perspective.

Meta-learning Primer. Meta-learning is known as learning to learn [26]. A good meta-learning model is capable of adapting well or generalizing to new environments that have never been encountered during training time. The adaptation process, essentially a mini-learning session (with limited exposure to the new environment), happens after the meta-learning model training stage. In the meta-learning model training stage, rather than training the learner on a single environment (with the goal of generalizing to unseen "intra-environment" samples from a similar data distribution), a meta-learner is trained on a distribution of environments, with the goal of learning a strategy that generalizes to unseen environments (i.e., "inter-environment"). Even without any explicit fine-tuning (i.e., with no gradient back-propagation on trainable variables), the meta-learning model autonomously adjusts internal hidden states to learn [12, 20, 39, 43].

Embedding Techniques. Embeddings map variables to low-dimensional vectors in a way that similar variables are close to each other [38, 57]. Embeddings have been widely used in the area of NLP and software engineering (e.g., word or code embeddings) and can also be applied to dense data to create a meaningful similarity metric. In AWARE, embeddings are used to explicitly represent and differentiate environments, and meta-learning enables learning to generate embeddings.

AWARE's Meta-learning Model Design. There are three key components in the design of the meta-learning model:

- A Distribution of MDPs (i.e., RL environments): Each MDP corresponds to one agent to which the base-learner will adapt. During the training of each agent, the meta-learner is exposed to a variety of environments and is trained to adapt to different MDPs. In our case of workload autoscaling, each environment represents a different application workload managed by the base-learner, where workloads can have heterogeneous SLOs, payloads, or architecture.
- A Model with Memory: We use a recurrent neural network (RNN) [17, 52, 55] that maintains a high-dimensional hidden state with nonlinear dynamics to acquire, process, and memorize knowledge about the current environment. In an RNN, hidden layers are recurrently used for computation. Compared to memoryless models such as autoregressive models and feed-forward neural networks, RNNs store information in the hidden states for a long time, so they are effective in capturing both spatial and temporal patterns. We did not explicitly use memory augmentation [51] for our RNN meta-learner because we found that the features of our application workloads are not as high-dimensional as those of computer vision tasks [51], and the RNN hidden states suffice to provide good representations.
- Meta-learning Algorithm: A meta-learning algorithm learns to update the base-learner to optimize for the purpose of adapting quickly to a previously unseen environment [20, 39]. Our novel approach uses an ordinary gradient descent update of RNN with a hidden state reset at a switch of MDPs. As training proceeds, the algorithm learns how to generate an embedding to best represent the environment and differentiate one environment from another.

Integration between Meta-learner and Base-learner. The base-learner discovers a rule that generalizes across data points for an <application, environment> pair, while the meta-learner generalizes across <application, environment> pairs. Fig. 5 illustrates the interaction between the meta-learner (❷ in Fig. 4) and the base-learner. Suppose that each data point used in the training and inference of the RL agent (i.e., a base-learner) with the <application, environment> pair i is $\{(s_t, a_t, r_t)\}_{0 \le t \le T}$, i.e., one RL trajectory TR_i from the environment i; then, each data point in the meta-learner is a bundle of M trajectories from the same environment, i.e., $[TR_1^i, TR_2^i, \ldots, TR_M^i]$. These episodes contain characteristics of the ongoing task that can be used to abstract some specific information about the environment (through <state, action, reward> transition sequences). The meta-learner uses a bidirectional RNN [52] to generate an embedding given a sequence of RL trajectories from the same environment (same base-learner). Unidirectional RNN has the limitation that it processes inputs in strict temporal order, so the current input has the context of previous inputs but not the future. Bidirectional RNN, on the other hand, duplicates the RNN processing chain so that the inputs are processed in both forward and backward orders to enable looking into future contexts as well.

The input trajectories (to the meta-learner) are selected from the RL trajectory database (that are generated by the RL agent interacting with the current RL environment) dynamically at runtime. We chose the top M trajectories that have resulted in the highest rewards so far because the experimental results show that the trajectories with lower rewards are unhelpful or even harmful. Intuitively, those lower-reward trajectories are generated with a random policy or a poorly trained policy, so they are not representative of the workloads.

The output from the bidirectional RNN of the meta-learner is an embedding that is used to fingerprint/represent the <application, environment> pair with which the base-learner is interacting. As shown in Fig. 5, the generated embedding based on past experience (i.e., the episodes previously explored by the base-learner) is fed to the base-learner as part of the input at each time step. Since we adopted as our base-learner the RL design from FIRM [47], which is an actor-critic RL algorithm, the embedding is taken by the actor network.

Interpreting Embeddings from Systems Perspective. The environment-specific embedding is able to differentiate one <application, environment> pair from another and thus guides the base-learner to adapt to the new environment. Fig. 6 visualizes the key idea of embedding. The spatial and temporal characteristics of the workloads are encoded and mapped onto a low-dimensional latent vector space by the embedding layer. Workloads with similar characteristics are projected to locations that are close to each other on that vector space. By calculating the cosine similarity between any two generated vectors (i.e., embeddings), we can get a monotonic similarity measure. To help understand how generated embeddings can represent spatial and temporal characteristics, we selected RL trajectories from <application, environment> pairs with human-detectable different performance sensitivities or load patterns, and then the plotted embedding projection shows that indeed similar workloads are closer to each other when comparing cosine similarities of their embeddings. In Fig. 6 (upper), the sensitivity of application performance to different resource allocations is shown in the heatmaps to illustrate the spatial characteristics, with the X-axis being CPU cores and the Y-axis being allocated RAM. Darker colors represent worse performance in terms of application request-serving latency. In Fig. 6 (lower), the application load-per-second time series are plotted to represent the temporal characteristics. Again, workloads with similar patterns are projected to adjacent locations in the output vector space.

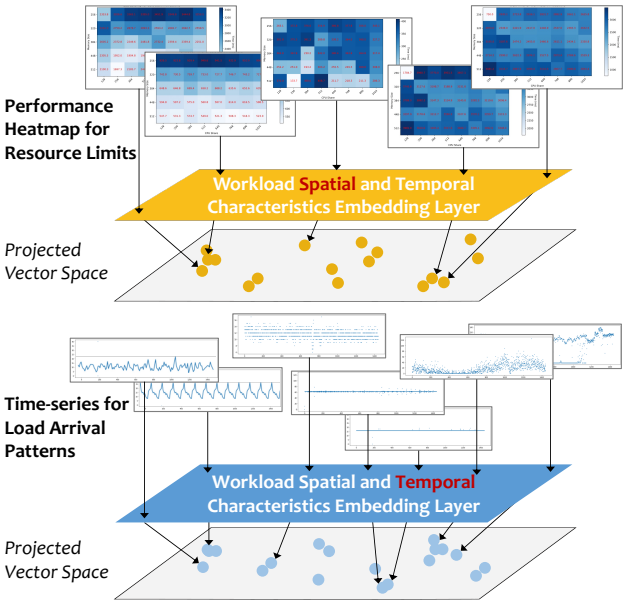

Figure 6: An example illustrating the idea of workload embedding for encoding spatial and temporal characteristics from a systems perspective. *The yellow points (upper figure) indicate that workloads with similar performance sensitivities (to resource allocations) are projected to locations near each other in the embedding vector space. The blue points (lower figure) show that workloads with similar load arrival patterns are projected to adjacent locations in the embedding vector space. The similarity metric used is cosine similarity.*

Meta-learner Training. During the training of the meta-learner, both the meta-learner and base-learner model parameters are updated. After each RL episode, the loss value is generated by the base-learner and is backward-propagated to update the model parameters in the base-learner. Since the meta-learner is trained across a distribution of environments, the total loss of all sampled environments in the training dataset is used to update the model parameters in the meta-learner. In the end, the trained meta-learner is capable of abstracting the individuality of each <application, environment> pair; the trained base-learner is a shared RL model that is able to generate optimal workload autoscaling policies conditioned on the workload embeddings provided by the meta-learner. The base-learner can be used as a starting point and as the basis for fine-tuning a specific novel <application, environment> pair in the inference stage.

Meta-learner Inference. After the meta-learner is trained, the meta-learning model is able to adapt the base-learner to a new <application, environment> pair that has never been encountered during training. Note that even though the new environment has never been encountered during training, it comes from the same distribution as, or shares similar patterns with, the encountered ones, so that transferring is still possible [12, 20, 39, 43]. The adaptation process only requires limited exposure to the new environment. Therefore, AWARE simply samples RL episodes and runs the meta-learner to generate the workload embedding. With the workload embedding, the base-learner can be continuously trained to learn the workload autoscaling policy for the new <application, environment> pair. The meta-learner model parameters are fixed during the inference stage.

Algorithm 1 RL agent lifecycle transition management for bootstrapping and triggering of online retraining. Four status codes `INITIALIZED`, `ONLINE`, `OFFLINE`, and `SERVING` stand for agent-initialized, online training, offline training, and online policy-serving, respectively.

Require: Rewards $R = [r_t]_{t \in T}$, User Profile P
1: **procedure** ObserveAndTrigger(R, P)
2: $stage \leftarrow$ INITIALIZED
3: **while** $True$ **do**
4: **if** $state$.equal(INITIALIZED) **then**
5: **if** P.BOOTSTRAP == $True$ **then**
6: $stage \leftarrow$ OFFLINE ▷ Bootstrapping
7: **else**
8: $stage \leftarrow$ ONLINE ▷ Skip Bootstrapping
9: **end if**
10: **else if** $state$.equal(OFFLINE) **then**
11: **if** avg(R) $\geq P.T_{online}$ **then**
12: $stage \leftarrow$ ONLINE
13: **end if**
14: **else if** $state$.equal(ONLINE) **then**
15: **if** avg(R) $\geq P.T_{serving}$ & std(R) $\leq P.T_{var}$ **then**
16: $stage \leftarrow$ SERVING
17: **end if**
18: **else if** $state$.equal(SERVING) **then**
19: **if** avg(R) $< P.T_{serving}$ $\|$ std(R) $> P.T_{var}$ **then**
20: $stage \leftarrow$ ONLINE
21: **end if**
22: **end if**
23: **end while**
24: **end procedure**

3.3 Incremental Retraining

When deploying the RL agent in a production system, one needs to ensure that the policy behaves as expected and scales to the workload in production. AWARE leverages continuous monitoring to detect any anomalous behavior and trigger retraining when needed. Alg. 1 describes how AWARE's RL retraining module (❸ in Fig. 4) manages the lifecycle of the agent and enables incremental retraining at runtime (lines 14–21). The input to the retraining module includes (a) the user profile specifying the configuration, and (b) recent rewards pulled from the RL trajectory database. When the mean and the standard deviation of the recent rewards satisfy the threshold-based condition (i.e., agent performance is bounded to a target value), the agent enters the policy-serving stage; otherwise, the agent enters the policy-training stage. Retraining of the RL agent is by online interaction with the RL environment. As discussed in §3.4, non-RL-based approaches

(i.e., HPA and VPA) can be used as a fallback option for RL agents when high-stakes applications want to keep the RL agent in the offline mode during retraining.

3.4 Bootstrapping

The policy at the early RL training stages could be worse than the baseline approaches. For example, overprovisioning leads to low resource utilization, while under-provisioning results in SLO violations. For production workloads, especially high-stakes applications, such suboptimal actions are not acceptable. In AWARE, an RL bootstrapper (❹ in Fig. 4) has been designed to combine offline and online RL training. If the user specifies enabling bootstrapping (as shown in lines 4–9 Alg. 1), the offline mode will be turned on first. AWARE will then use Kubernetes HPA [25] (which is a threshold-based approach) for horizontal workload autoscaling, and use Kubernetes VPA [15] (which adjusts resource limits based on history profile) for vertical workload autoscaling. Note that HPA and VPA can also be used as a fallback option for RL when high-stakes applications want to keep the RL agent in the offline mode during retraining, as discussed in §3.3.

In the offline mode, the RL bootstrapper intercepts the request-response path between the agent and the RL API gateway and replaces the RL agent with the fallback controller to react to the received states and generate actions at each time step. The RL API gateway then takes the received action for execution, and the resulting behavior is the same as when workloads are managed by HPA and VPA. The RL agent samples trajectories from the trajectory database for offline policy training. To overcome extrapolation errors whereby previously unseen state-action pairs are erroneously estimated, we apply a state-conditioned generative model to combine with the critic network for producing previously seen actions [14]. In the online training mode, the agent will then directly interact with the RL environment through the API gateway.

4 Implementation

4.1 Kubernetes MPA

We propose our own design and implementation of multi-dimensional Pod autoscaling because the current HPA and VPA controllers are independent of each other and can lead to a large number of tiny Pods [16]. Google MPA [10] is a pre-GA beta version product that offers an integrated solution for HPA and VPA, but it is not open-sourced and does not support custom recommenders. In AWARE, the MPA framework combines the actions of vertical and horizontal autoscaling but separates the actuation from the controlling algorithms. As shown in Fig. 7, there are three controllers (i.e., a recommender, an updater, and an admission controller) and an MPA API (i.e., a CRD object [24]) that connects the autoscaling recommendations to actuation.

The multidimensional scaling algorithm is implemented in the recommender mostly by importing HPA and VPA libraries to serve as the fallback option for RL-based approaches. The

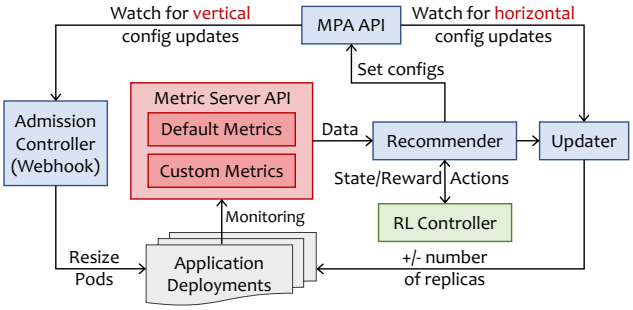

Figure 7: MPA design overview and integration with RL.

metrics required by the algorithm are collected from the Kubernetes Metrics Server, including default metrics such as container resource utilizations and custom metrics such as application throughput or latency. The scaling decisions derived from the recommender are stored in the MPA object as scaling configurations. The updater and the admission controller retrieve those updated configurations from the MPA object and then actuate them as vertical and horizontal actions on the application Deployments. The separation of action actuation from scaling decision generation allows developers to replace the default recommender with the alternative recommender, i.e., the RL controller. The implementation is in Go and at the stage of releasing to the Kubernetes upstream as well.

4.2 Integration with RL

The creation of MPA is through declarative YAML files. To integrate MPA with RL agents, one needs to specify a custom recommender to replace the default recommender (HPA+VPA). After an MPA is initialized for the application deployment, an MPA wrapper is created as a shim layer to communicate with the RL agent through RPCs. We follow the "agent-centric" pattern of request-response interaction advocated by OpenAI Gym [44]. The exposed interfaces include (a) `init()` (for initializing the RL environment), (b) `state = reset()` (for resetting the environment at the beginning of each RL episode), and (c) `state, reward = step(action)` (for RL agent stepping). When the MPA wrapper receives an action through the RPC request, it first translates the action to vertical and horizontal scaling configurations and writes to the MPA object. We deploy Prometheus [7], the standard monitoring service in Kubernetes, to export default and custom metrics from the application Deployment. The wrapper then queries the Prometheus service for real-time metrics and translates to RL states and rewards. Finally, the wrapper sends the metrics back to the agent through the RPC response. The MPA wrapper is implemented in Python.

4.3 Meta-learning-based RL-serving

AWARE's meta-learning-based RL agent management framework is implemented in Python. Both the base-learner (adopted from FIRM [47]) and the meta-learner are implemented using PyTorch [13]. The meta-learner is essentially a bidirectional two-layer RNN followed by two fully connected

layers with the ReLU activation function. We chose the trajectory bundle size to be 20 for the fastest adaptation with the fewest trajectories according to the sensitivity analysis. Each RNN hidden layer consists of 256 neurons, and the fully connected layers consist of 256 and 64 neurons. We chose two layers and an embedding size of 64 because adding more layers and hidden units does not increase performance in our experiments; instead, it slows down training speed significantly. We used the Adam optimizer for parameter updates.

RL trajectories are saved to InfluxDB [21], an open-source time-series database that is built to handle metrics with time-stamped data. Recent rewards, sampled RL trajectories for offline base-learner training, and the inputs for embedding generation are all pulled from the trajectory database by using the InfluxDB Python client library.

AWARE provides a simple and declarative user interface for RL pipeline developers, which is consistent with Kubernetes' way of creating and managing objects in the cluster. To specify the targets for the workloads, i.e., resource utilization targets and the application SLO (if there is one), users only need to provide a YAML file following the definition template. Both application latency and throughput SLOs are currently supported. In addition, users can also specify the thresholds for RL rewards and whether or not to enable bootstrapping in the YAML file, which constructs the profile used in Alg. 1.

5 Evaluation

Our experiments addressed the following research questions:

§5.2 Does AWARE provide fast model adaptation to new workloads? What is the value of meta-learning?

§5.3 How does AWARE perform in online policy-serving when workload updates or load changes occur?

§5.4 How does AWARE perform in the early stages of policy training, compared to RL agents without bootstrapping?

5.1 Experimental Setup

We implemented an application generator capable of generating a large number of synthetic applications by combining the 16 selected representative production application segments [11] (discussed in §2.3 as well) based on random sampling with replacement from the segment pool. Each segment represents the smallest granularity of common workloads in cloud datacenters. In addition, each segment has to be associated with its own inputs to simplify load generation (e.g., the image manipulation workloads come with random images). The generator also comes with setup and tear-down scripts for all external services each segment uses (e.g., databases or messaging queues). Overall, we generated 1000 unique applications, deployed them as Deployments in a Kubernetes cluster of 11 two-socket physical nodes, and ran an RL-based multidimensional autoscaler with each application. Each server consists of 56–192 CPU cores and RAM that vary from 500 GB to 1000 GB. Seven of the servers use Intel x86 Xeon E5 processors, and the remaining ones use IBM ppc64 Power8 and Power9 processors.

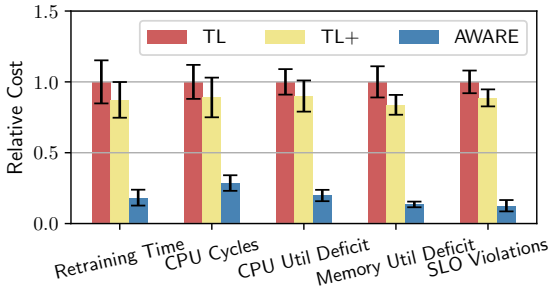

Figure 8: RL agent retraining cost and performance comparison of AWARE, transfer learning (TL), and transfer learning with augmented features (TL+).

While it would be impossible to cover all cloud workloads, the selected production workload segments should enable the generation of a large number of synthetic cloud workloads with varying resource consumption profiles. In the future, the number of implemented segments can easily be extended if specific workload profiles are missing. We refer to the open-source artifact for additional details on the generator implementation. With the same datacenter workload traces [65] discussed in §2.3 with respect to RL agent training and policy-serving, we divided the 1000 generated application pool with the 8:2 ratio. The 800 applications with varied workloads are used to train the meta-learner, while the remaining 200 applications are used to evaluate the adaptability. The total runtime is ∼60 days, and the meta-learner training time is ∼312 hours on an Intel(R) Xeon(R) E5-2695 processor.

The RL formulation and design (in the base-learner) are adopted from FIRM [47] (as mentioned in §2.3). As an end-to-end evaluation, Fig. 1 shows that, compared to AWARE, the RL-based autoscaler FIRM by itself suffered from poor performance during the initial training stage (i.e., Stage ①, which demonstrates the benefit of AWARE's bootstrapping mechanism), online policy-serving performance degradation (i.e., Stage ②, which demonstrates the benefit of the online retraining triggering mechanism), and slow adaptation with non-trivial retraining (i.e., Stage ③, which demonstrates the benefit of meta-learning). We then present the evaluation results related to each research question in §5.2–§5.4.

5.2 Fast Adaptation

To study adaptability to new workloads, we compared AWARE with the existing transfer learning approach. FIRM [47] leverages transfer learning to train an RL agent for a new service based on previous RL experience gained when training the RL agent for a known service. In the transfer-learning-based approach (TL), the model parameters (weights) are shared between the agents managing the known workload and the new workload. We also compared AWARE with a novel approach (TL+) based on transfer learning that includes additional spatial and temporal features in the RL states, since the meta-learner in AWARE is trained to output an embedding to represent the spatial and temporal characteristics of the <application, environment> pair. We

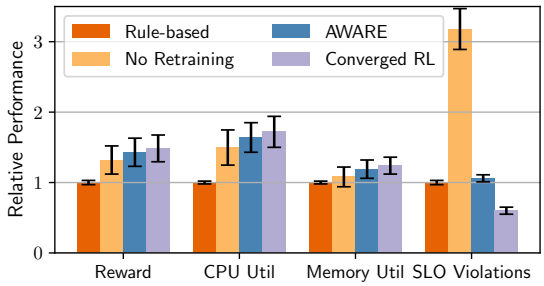

Figure 9: RL agent online policy-serving performance comparison of AWARE, no retraining, the rule-based method, and the agent with the converged RL policy. *In the comparison of reward and CPU/memory utilization, the higher, the better, while a lower number of SLO violations is better.*

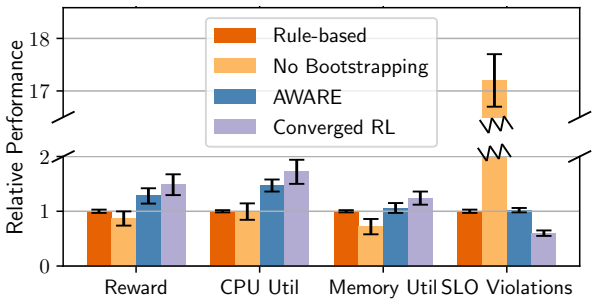

Figure 10: RL agent training performance comparison of AWARE, no bootstrapping, the rule-based method, and the agent with the converged RL policy. *In the comparison of reward and CPU/memory utilization, the higher, the better, while a lower number of SLO violations is better.*

used the widely used ARIMA model [19] to generate the predicted load for the next time step (i.e., temporal feature) and recorded a table mapping from resource allocation to performance (i.e., spatial feature). We performed A/B tests in which the workload traces were the same, but the recommender in MPA was replaced with TL, TL+, and AWARE, which drove the horizontal and vertical scaling of the workload. We repeated the A/B test 100 times. In each test, we randomly selected a workload from the pool and trained the RL agent to convergence. We then randomly selected 10 other different workloads from the pool for adaptivity evaluation. We measured the retraining time, CPU cycles involved in retraining, utilization deficit (compared to the converged RL policy), and SLO violations.

Fig. 8 shows that AWARE adapted 5.5× and 4.6× faster (saved 68–72% CPU cycles) than TL and TL+, respectively. During the adaptation period, TL+ had 4.6× and 6.2× higher CPU and memory utilization deficit compared to AWARE while AWARE reduced SLO violations by 7.1×. TL+ encodes additional spatial and temporal features, but each state is still a stateless snapshot of the running workload. Additional features (i.e., the table and the ARIMA output) greatly increase the state space. Meta-learning, on the other hand, offers a systematic and automated way of learning how to differentiate the workloads well and outputs a low-dimensional embedding to be used by the base-learner.

5.3 Online Policy-serving

To evaluate the online policy-serving performance when facing workload updates and load changes (described in §2.3), we compared AWARE with (a) a rule-based approach, (b) an RL agent without continuous monitoring and retraining, and (c) an RL agent with the converged policy. For the rule-based approach with manual scaling, we measured the maximum CPU utilization when the SLO was met, and set it as the threshold for HPA. We used the default Auto mode [15] for VPA. We performed the same style of A/B tests 100 times and replaced the MPA recommender with the four approaches. In each A/B test, we randomly selected a workload from the pool, trained the RL agent to convergence, and injected a series of the seven random instability scenarios introduced in §2.3. We then measured the average reward, CPU/memory utilization, and the number of SLO violations during the time until the agent managed by AWARE converged. Fig. 9 shows that AWARE had 9.6% and 14.8% higher CPU and memory utilization, and reduced SLO violations by 3.1× compared to the RL agent without retraining (the second-best approach), resulting in 8.6% higher per-episode reward. Compared to the converged RL policy, AWARE had a 3.6% lower average per-episode reward because we set the retraining threshold to be 5, corresponding to a 2.6% reward degradation. Sensitivity analysis showed that AWARE converged to the no-retraining baseline as the threshold increased while a smaller than 5 threshold led to constant retraining with no policy serving.

5.4 Bootstrapping

To study how much bootstrapping helps reduce the cost of early-stage RL training, we compared AWARE with (a) the rule-based approach (same as in Fig. 9), (b) an RL agent without bootstrapping, and (c) an RL agent with the converged policy. Since the per-episode reward achieved by the rule-based autoscaler is around 130 (translated from the measured utilization and performance), we set the bootstrapping threshold in AWARE to 130. In the sensitivity analysis, we observed that a higher threshold led to endless bootstrapping driven by the rule-based autoscaler (since the measured reward is always lower than the threshold), while the lower the threshold, the more performance degradation AWARE had during its early-stage training. A threshold of 0 basically converges to the learning curve without AWARE bootstrapping (i.e., no offline learning). We performed A/B tests 100 times and replaced the MPA recommender with the four approaches. In each A/B test, we randomly selected a workload from the pool and trained the RL agent to convergence (with or without bootstrapping). We then measured the average reward, CPU utilization, memory utilization, and the number of SLO violations during the time until the RL agent converged. Fig. 10 shows that AWARE had 47.5% and 39.2% higher CPU and memory utilization, respectively, and reduced SLO viola-

tions by a factor of 16.9× compared to the RL agent without bootstrapping (the second-best approach), resulting in 47.3% higher average per-episode reward before convergence.

6 Related Work

RL Training and Model-serving Frameworks. Ray [41] is an open-source distributed execution framework that facilitates RL model training and serving by making it easy to scale an RL application and schedule distributed runs to efficiently use all resources (i.e., CPU, memory, or GPU) available in a cluster. Amazon SageMaker [1] uses the Ray RLlib library that builds on the Ray framework to train RL policies. SageMaker also provides cloud services that help build and deploy ML models (e.g., data processing and model evaluation). RLzoo [9] is an RL library that aims to make the development of RL agents efficient by providing high-level yet flexible APIs for prototyping RL agents. RLzoo also allows users to import a wide range of RL agents and easily compare their performance. Park [33] provides 12 representative RL environments in the field of systems and networking (e.g., job scheduling) for developing and evaluating RL algorithms. Genet [60] is an RL training framework for learning better network adaptation policies. Genet leverages curriculum learning [42], which aims to sequence tasks to achieve the best performance on a specific final task instead of quickly adapting to a new task within a small number of gradient descent steps.

RL in Production. Panzer *et al.* [45] provide a survey of existing RL applications in production system domains, including resource scheduling. They summarize the implementation challenges and generalizability of simulation-trained RL models. SOL [58] is an extensible framework for developing ML/RL-based controllers for tasks such as core frequency scaling. SOL is complementary to AWARE, which can further guarantee that the RL agent operates safely under various realistic issues, including bad data and external interference like resource unavailability. SIMPPO [36, 48] provides a scalable framework based on the mean-field theory that enables multiple RL agents to coexist in a shared multi-tenant environment. Autopilot [50] is a workload autoscaler used at Google that leverages multi-armed bandits (i.e., the simplest version of RL) to choose a variant of the sliding window algorithms that historically would have resulted in the best performance for each job. In its essence, it is still a heuristic mechanism and has been shown [59] to suffer from poor system stability because of inaccurate estimation of horizontal concurrency; it can also result in a large number of tiny Pods [16] due to the independence between horizontal and vertical scaling.

7 Discussion and Future Challenges

Extension to Other System Domains. AWARE is a general and extensible framework that can be applied to other systems management tasks (e.g., congestion control or job scheduling). To apply it to a new domain, one needs to (a) replace the RL environment by implementing the provided environment wrapper interface; and (b) provide a default non-RL-based agent for the RL bootstrapper. We leave the study of the performance in other system domains to the future.

Out-of-distribution Workloads. AWARE provides the opportunity to quickly customize the model to specific workloads. However, out-of-distribution <application, environment> pairs still require training because meta-learning assumes that all pairs, including the unseen cases, are inherently within the learned distribution [20] (e.g., in terms of service request arrival patterns or sensitivity to resource allocation). Given the diversity of workloads in the cloud datacenter (used in the training dataset), the meta-learner and the shared base-learner can be continuously trained, and out-of-distribution cases are covered eventually. Meanwhile, with offline RL training, users can still benefit from the heuristics-based solution used as the fallback option. One limitation of our experiment was that the generated applications might not have covered all possible cloud workloads. However, application segments can easily be extended in the synthetic application generator if specific workload profiles are missing (§5.1).

On-policy RL Algorithms. When RL agents are being bootstrapped at the initial stage, off-policy RL agents (such as DDPG [29, 47] and DQN [59, 62]) can be trained directly using the collected RL trajectories. However, on-policy RL agents (such as PPO [48]) require trajectories generated from their own policy. One potential way to train on-policy RL agents offline would be to build a simulator based on the collected trajectories, which would essentially map resource allocation to workload performance and system metrics. A balanced experience replay scheme [27] could potentially be applied for locating near-on-policy samples from the simulator constructed based on the offline dataset. Instead of drawing trajectories from the trajectory database (as in §3.4), the RL base-learner can interact with the simulator for bootstrapping.

8 Conclusion

This paper explored the challenges of applying RL in workload autoscaling in production cloud platforms. We presented a general and extensible framework for deploying and managing RL agents in production systems. To demonstrate the framework, we implemented AWARE for automating RL-based workload autoscaling in Kubernetes and experimentally showed (a) the benefits of leveraging meta-learning for fast model adaptation, and (b) how the design of AWARE ensures the stable and robust online performance of RL models.

Acknowledgments

We thank the anonymous reviewers and our shepherd Xiaosong Ma for their valuable comments that improved the paper. This work is partially supported by the National Science Foundation (NSF) under grant No. CCF 20-29049; and by the IBM-ILLINOIS Discovery Accelerator Institute (IIDAI). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or IBM.

Availability

We provide an open-source implementation of AWARE at https://gitlab.engr.illinois.edu/DEPEND/aware.

References

[1] AWS. Amazon SageMaker. https://aws.amazon.com/sagemaker/, 2022. Accessed: 2022-11-23.

[2] AWS. AWS autoscaling documentation. https://docs.aws.amazon.com/autoscaling/index.html, 2022. Accessed: 2022-11-23.

[3] Azure. Azure autoscale. https://azure.microsoft.com/en-us/features/autoscale/, 2022. Accessed: 2022-11-23.

[4] Subho Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Inductive-bias-driven reinforcement learning for efficient scheduling in heterogeneous clusters. In *Proceedings of the 37th International Conference on Machine Learning (ICML 2020)*, pages 629–641, Cambridge, MA, USA, 2020. PMLR.

[5] Jingde Chen, Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Machine learning for load balancing in the Linux kernel. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (ApSys 2020)*, pages 67–74, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Google Cloud. Google cloud load balancing and autoscaling. https://cloud.google.com/compute/docs/load-balancing-and-autoscaling, 2022. Accessed: 2022-11-23.

[7] CNCF. Prometheus. https://prometheus.io/, 2022. Accessed: 2022-11-23.

[8] Sundar Dev, David Lo, Liqun Cheng, and Parthasarathy Ranganathan. Autonomous warehouse-scale computers. In *ACM/IEEE 57th Design Automation Conference (DAC 2020)*, pages 1–6, 2020.

[9] Zihan Ding, Tianyang Yu, Hongming Zhang, Yanhua Huang, Guo Li, Quancheng Guo, Luo Mai, and Hao Dong. Efficient reinforcement learning development with RLzoo. In *Proceedings of the 29th ACM International Conference on Multimedia (MM 2021)*, pages 3759–3762, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Google GKE Documentation. Configuring multidimensional Pod autoscaling in GKE. https://cloud.google.com/kubernetes-engine/docs/how-to/multidimensional-pod-autoscaling, 2022. Accessed: 2022-11-23.

[11] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39, 2021.

[12] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, pages 1126–1135. JMLR.org, 2017.

[13] The Pytorch Foundation. PyTorch. https://pytorch.org/, 2022. Accessed: 2022-11-23.

[14] Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. In *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*, pages 2052–2062. PMLR, 2019.

[15] GitHub. Vertical Pod Autoscaling (VPA) in Kubernetes. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler, 2022. Accessed: 2022-11-23.

[16] Google GKE. Challenges of scaling kubernetes Pods horizontally and vertically. https://cloud.google.com/blog/topics/developers-practitioners/scaling-workloads-across-multiple-dimensions-gke, 2022. Accessed: 2022-11-23.

[17] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[18] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*, pages 620–629, 2018.

[19] Siu Lau Ho and Min Xie. The use of ARIMA models for reliability forecasting and analysis. *Computers & Industrial Engineering*, 35(1-2):213–216, 1998.

[20] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey. Meta-learning in neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(09):5149–5169, 2022.

[21] InfluxData. InfluxDB. https://github.com/influxdata/influxdb, 2022. Accessed: 2022-11-23.

[22] Nathan Jay, Noga H. Rotman, P. Godfrey, Michael Schapira, and Aviv Tamar. Internet congestion control via deep reinforcement learning. *In Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS 2018)*, 32, 2018.

[23] Sara Kardani-Moghaddam, Rajkumar Buyya, and Kotagiri Ramamohanarao. ADRL: A hybrid anomaly-aware deep reinforcement learning-based resource scaling in clouds. *IEEE Transactions on Parallel and Distributed Systems (TPDS 2020)*, 32(3):514–526, 2020.

[24] Kubernetes. Extending Kubernetes API with custom resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources, 2022. Accessed: 2022-11-23.

[25] Kubernetes. Horizontal Pod Autoscaling (HPA) in Kubernetes. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2022. Accessed: 2022-11-23.

[26] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[27] Seunghyun Lee, Younggyo Seo, Kimin Lee, Pieter Abbeel, and Jinwoo Shin. Offline-to-online reinforcement learning via balanced replay and pessimistic Q-ensemble. In *Proceedings of the 5th Conference on Robot Learning (CoRL 2021)*, pages 1702–1712. PMLR, 2021.

[28] Xu Li, Feilong Tang, Jiacheng Liu, Laurence T. Yang, Luoyi Fu, and Long Chen. AUTO: Adaptive congestion control based on multi-objective reinforcement learning for the satellite-ground integrated network. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC 2021)*, pages 611–624. USENIX Association, 2021.

[29] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *Proceedings of the 4th International Conference on Learning Representations (ICLR 2016)*, 2016. https://arxiv.org/abs/1509.02971.

[30] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA 2015)*, pages 450–462, 2015.

[31] Yiqing Ma, Han Tian, Xudong Liao, Junxue Zhang, Weiyan Wang, Kai Chen, and Xin Jin. Multi-objective congestion control. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys 2022)*, EuroSys '22, pages 218–235, New York, NY, USA, 2022. Association for Computing Machinery.

[32] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNet 2016)*, pages 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, et al. Park: An open platform for learning-augmented computer systems. *Advances in Neural Information Processing Systems (NeurIPS 2019)*, 32, 2019. https://proceedings.neurips.cc/paper/2019.

[34] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2017)*, pages 197–210, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM 2019)*, pages 270–288, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Weichao Mao, Haoran Qiu, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, and Tamer Başar. A mean-field game approach to cloud resource management with function approximation. In *Proceedings of the 36th Conference on Advances in Neural Information Processing Systems (NeurIPS 2022)*, volume 36, pages 1–12, New Orleans, LA, USA, 2022. Curran Associates, Inc.

[37] Jason Mars and Lingjia Tang. Whare-Map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA 2013)*, pages 619–630, New York, NY, USA, 2013. Association for Computing Machinery.

[38] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural*

Information Processing Systems (NeurIPS 2013), pages 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc.

[39] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. In *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*, 2018. https://openreview.net/forum?id=B1DmUzWAW.

[40] Jun Morimoto and Kenji Doya. Robust reinforcement learning. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems (NeurIPS 2000)*, volume 13. MIT Press, 2000. https://proceedings.neurips.cc/paper/2000.

[41] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI 2018)*, pages 561–577, USA, 2018. USENIX Association.

[42] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research*, 21(1), 2022. https://jmlr.org/papers/volume21/20-212/20-212.pdf.

[43] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

[44] OpenAI. OpenAI Gym documentation. https://www.gymlibrary.dev/, 2022. Accessed: 2022-11-23.

[45] Marcel Panzer and Benedict Bender. Deep reinforcement learning in production systems: A systematic literature review. *International Journal of Production Research*, 60(13):4316–4341, 2022.

[46] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. Robust adversarial reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, pages 2817–2826. JMLR.org, 2017.

[47] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, pages 805–825, Berkeley, CA, USA, November 2020. USENIX Association.

[48] Haoran Qiu, Weichao Mao, Archit Patke, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. SIMPPO: A scalable and incremental online learning framework for serverless resource management. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC 2022)*, pages 306–322, New York, NY, USA, 2022. Association for Computing Machinery.

[49] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *Proceedings of the 12th International Conference on Cloud Computing (CLOUD 2019)*, pages 329–338, 2019.

[50] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at Google. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys 2020)*, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3342195.3387524.

[51] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML 2015)*, pages 1842–1850. JMLR.org, 2016.

[52] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[53] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)*, pages 733–750, New York, NY, USA, 2020. Association for Computing Machinery.

[54] Ion Stoica and Scott Shenker. From cloud computing to sky computing. In *The 18th Workshop on Hot Topics in Operating Systems (HotOS 2021)*, pages 26–32, New York, NY, USA, 2021. Association for Computing Machinery.

[55] Ilya Sutskever, James Martens, and Geoffrey E. Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML 2011)*, 2011. https://icml.cc/2011/papers/524_icmlpaper.pdf.

[56] Chen Tessler, Yuval Shpigelman, Gal Dalal, Amit Mandelbaum, Doron Haritan Kazakov, Benjamin Fuhrer, Gal Chechik, and Shie Mannor. Reinforcement learning for datacenter congestion control. *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 2022)*, 36(11):12615–12621, Jun. 2022.

[57] Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: A simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL 2010)*, pages 384–394, USA, 2010. Association for Computational Linguistics.

[58] Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. SOL: Safe on-node learning in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*, pages 622–634, New York, NY, USA, 2022. Association for Computing Machinery.

[59] Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, K. K. Ramakrishnan, Yangfei Zheng, Meng Yan, Xiaohong Zhang, and Alex X. Liu. DeepScaling: Microservices autoscaling for stable CPU utilization in large scale cloud systems. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC 2022)*, pages 16–30, New York, NY, USA, 2022. Association for Computing Machinery.

[60] Zhengxu Xia, Yajie Zhou, Francis Y. Yan, and Junchen Jiang. Genet: Automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 397–413, New York, NY, USA, 2022. Association for Computing Machinery.

[61] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: A randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2020)*, pages 495–511, 2020.

[62] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *IEEE 39th International Conference on Distributed Computing Systems (ICDCS 2019)*, pages 122–132, Washington, DC, USA, 2019. IEEE Computer Society.

[63] Hanfei Yu, Athirai A. Irissappane, Hao Wang, and Wes J. Lloyd. FaaSRank: Learning to schedule functions in serverless platforms. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS 2021)*, pages 31–40, Washington, DC, USA, 2021. IEEE Computer Society.

[64] Kuo Zhang, Peijian Wang, Ning Gu, and Thu D. Nguyen. GreenDRL: Managing green datacenters using deep reinforcement learning. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC 2022)*, pages 445–460, New York, NY, USA, 2022. Association for Computing Machinery.

[65] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021)*, pages 724–739, New York, NY, USA, 2021. Association for Computing Machinery.