FLASH: FAST MODEL ADAPTATION IN ML-CENTRIC CLOUD PLATFORMS

Haoran Qiu [1] **Weichao Mao** [2] **Archit Patke** [2] **Shengkun Cui** [2] **Chen Wang** [3] **Hubertus Franke** [3]
Zbigniew T. Kalbarczyk [2] **Tamer Başar** [2] **Ravishankar K. Iyer** [1 2]

ABSTRACT

The emergence of ML in various cloud system management tasks (e.g., workload autoscaling and job scheduling) has become a core driver of *ML-centric cloud platforms*. However, there are still numerous algorithmic and systems challenges that prevent ML-centric cloud platforms from being production-ready. In this paper, we focus on the challenges of model performance variability and costly model retraining, introduced by dynamic workload patterns and heterogeneous applications and infrastructures in cloud environments. To address these challenges, we present FLASH, an extensible framework for fast model adaptation in ML-based system management tasks. We show how FLASH leverages existing ML agents and their training data to learn to generalize across applications/environments with meta-learning. FLASH can be easily integrated with an existing ML-based system management agent with a unified API. We demonstrate the use of FLASH by implementing three existing ML agents that manage (1) resource configurations, (2) autoscaling, and (3) server power. Our experiments show that FLASH enables fast adaptation to new, previously unseen applications/environments (e.g., 5.5× faster than transfer learning in the autoscaling task), indicating significant potential for adopting ML-centric cloud platforms in production.

1 INTRODUCTION

ML techniques such as supervised learning (SL) and reinforcement learning (RL) have been widely applied in many system management tasks, e.g., resource management (Qiu et al., 2020), job scheduling (Mao et al., 2019b), and power management (Wang et al., 2022a). However, from production experience at Microsoft (Liang et al., 2020), costly model retraining (regarding computation time, energy consumption, and additional hand-made data collection) is necessary to adapt to previously unseen applications or deployment environments that are constantly introduced in heterogeneous (or even multi-cloud) datacenters (Mars & Tang, 2013; Stoica & Shenker, 2021).

In this paper, we aim to facilitate efficient ML model adaptation in practice. The goal is ***not*** to revise existing ML/RL algorithms or modeling approaches that have been proposed to handle various system management tasks but to improve existing model training and adaptation in a *transparent* manner. We introduce **FLASH**, a general framework that assists developers in training and deploying ML agents with fast adaptability to diverse cloud applications and environments in the context of different system management tasks.

Background. FLASH is in line with the vision of an *ML-centric cloud platform* (Bianchini et al., 2020) that embeds ML to handle various system management tasks throughout the platform by learning from data and automating management decisions. As shown in Figure 1, a system management ***task*** (e.g., workload autoscaling in a Kubernetes cluster) involves three main components: ***application*** (e.g., the deployed workload), ***environment*** (e.g., the underlying infrastructure), and ***management agent*** (i.e., either an ML agent with a model trained to perform the task or a non-ML agent based on static heuristics developed with offline profiling). Independently for each task, FLASH reduces the model retraining cost when adapting the ML agent across applications and environments specific to that task.

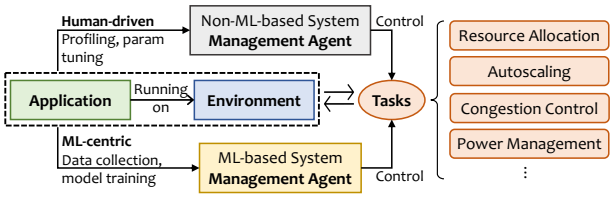


Figure 1. An ML-centric cloud platform with a mix of non-ML- and ML-based system management agents.

Our Approach. FLASH is the first general framework that introduces *embedding-based meta-learning* (Mishra et al., 2018; Rusu et al., 2019) for ML-based cloud system management. Given a management task (e.g., workload autoscaling) and an ML agent (originally designed and trained to handle that task in the context of application A_i and environment E_j), FLASH introduces an abstraction of a ***base learner***. FLASH's objective is to facilitate fast adaptation (i.e., reduced training time) of the base learner to novel (A_i, E_j)

[1]Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, IL [2]Department of Electrical and Computer Engineering, University of Illinois Urbana-Champaign, Urbana, IL [3]IBM Research, Yorktown Heights, NY.

Proceedings of the 7^{th} MLSys Conference, Santa Clara, CA, USA, 2024.

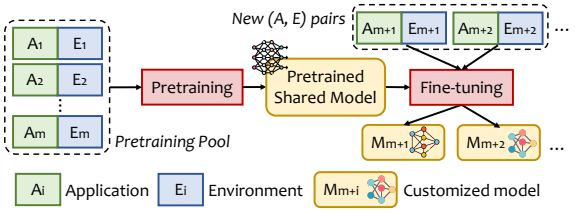


Figure 2. Pretrain-finetune workflow in FLASH with the abstraction of application as A_i, environment as E_i, and model as M_i in each system management task.

pairs. Toward this goal, we introduce a ***meta learner*** trained to handle a given management task and generalize across different (A_i, E_j) pairs. The meta learner captures application and environment heterogeneity by explicitly modeling the unique characteristics of (A_i, E_j) pairs using learned *embedding representations*, and thus benefits adaptation.

FLASH operates in two phases: *pretraining* and *fine-tuning*, as shown in Figure 2. Instead of training a model for each (A_i, E_j) pair, FLASH first pre-trains a shared model as the common basis with *meta-learning*. To rapidly adapt to a new (A_i, E_j) pair in a task, the shared model is then further fine-tuned by conditioning on the embedding generated (by the meta learner) for the new (A_i, E_j) pair.

The core design of FLASH originates from this key insight that *existing training datasets of an ML agent inherently contain spatial-temporal characteristics specific to each (A_i, E_j) pair.* However, those characteristics are poorly utilized to generalize across (A_i, E_j) pairs because the ML agent's goal is to specialize for the specific pair that the agent is lastly trained on. For instance, in resource management, workload's performance sensitivities to resource configurations (i.e., *spatial* characteristics) can be derived from the labels in supervised learning (Eismann et al., 2021a) or RL rewards (Qiu et al., 2020). In autoscaling, the time-varying load patterns (i.e., *temporal* characteristics) are available in RL trajectories (Mao et al., 2019a). To utilize available datasets and learn to generalize across (A_i, E_j) pairs, we leverage ***meta-learning*** (Hospedales et al., 2022), a family of techniques known as "learning to learn" (Thrun & Pratt, 1998), which has demonstrated fast model adaptivity in image classification (Rusu et al., 2019), robotics (Mishra et al., 2018), and cybersecurity (Yang et al., 2023).

Use Cases. We demonstrate the use of FLASH by integrating it with three independent ML-based system management agents (as shown in Table 1): (1) a resource configuration agent (i.e., Sizeless (Eismann et al., 2021a)) that predicts the performance of a serverless function on various resource configurations, (2) a workload autoscaling agent (i.e., from FIRM (Qiu et al., 2020)) that scales workload resources to achieve application service-level objectives (SLOs) with high resource utilization, and (3) a power management agent (i.e., SmartOverclock (Wang et al., 2022a)) that overclocks CPU cores (by scaling up frequency) only when it benefits the workloads. All three agents are handled by FLASH in a unified manner *without any changes to the original model design*, regardless of the modeling approach (i.e., supervised learning or RL), architecture, or optimization constraints.

Results. We present a detailed experimental evaluation of our agents and show that FLASH provides fast model adaptation with lightweight overhead. As an example, to retrain an RL-based autoscaler (i.e., FIRM model) to convergence for new applications or environments, FLASH is 5.5× faster than the state-of-the-art transfer learning-based approach used in FIRM. Even without retraining, the FIRM agent integrated with FLASH has 71.6% less performance degradation (regarding RL reward) when testing trained models on new applications than the original FIRM agent (reduction from 37% reward drop to 10.5%) as we show in §5.2.

Contributions. Our main contributions are:

- The design of FLASH, the first training framework that systematically introduces *meta-learning* and a *pretrain-finetune* paradigm to ML-based cloud system management tasks in a unified manner that can rapidly adapt to new or updated applications or environments.
- A characterization of the model adaptation cost when adapting three existing ML-based system management agents across various applications and environments.
- Demonstration of FLASH with the three representative agents showing substantial improvements in adaptation.
- A *similarity metric* to infer performance degradation when serving ML models for new applications or environments and the retraining overhead during adaptation.

Putting FLASH in Perspective. To achieve optimal performance customized to any (A_i, E_j) pair, one approach is to train one model per pair (e.g., (Qiu et al., 2020; Zhang et al., 2021c)), but it leads to significant overhead in maintaining all individualized models. To reduce the number of models, one can train a shared model with a unique ID assigned to each application (e.g., (Wang et al., 2022b)), but it is not scalable to the growing (A_i, E_j) spaces, and requires retraining the whole model for any updated or newly added (A_i, E_j) pairs. While state-of-the-art transfer-learning-based approaches are useful in reducing the model adaptation cost (Qiu et al., 2020), they suffer from the trade-off between the adaptation cost and model management overhead (by keeping individual models and identifying the best one to transfer from). In the systems domain, it is challenging to classify applications/environments into groups at an optimal granularity (with careful feature selection) and then train one model per group to serve as the transfer basis.

FLASH, in contrast, provides a systematic and automatic way of adapting the model to a new (A_i, E_j) pair based on the similarity of the embedding representation of the new pair with the seen (A_i, E_j) pairs. In addition, FLASH only requires maintaining a single pre-trained shared model for all (A_i, E_j) pairs while supporting lightweight fine-tuning.

Table 1. FLASH use cases and ML agents developed for cloud system management tasks. See §4 for detailed descriptions.

Use Case	**Application**	**Environment**	**Agent**	**Model**	**Input / State** (RL)	**Output / Action** (RL)	**Loss / Reward** (RL)
Resource Config Search (RCS) (Eismann et al., 2021a)	Serverless Function	Serverless Platform	SL	Fully Connected Neural Network	Base memory, Execution time, Target memory, Node.js `process.resourceUsage()` and `process.cpuUsage()` outputs	Execution time	Mean absolute percentage error (MAPE)
Workload Autoscaling (WA) (Qiu et al., 2020)	Kubernetes Deployment	Kubernetes Cluster	RL	DDPG (Deep Deterministic Policy Gradient)	Resource limits (CPU, RAM), Resource utilization (RU), SLO preservation ratio (SP), Load changes	Resource limits, Number of replicas	$\alpha \cdot SP_t + (1-\alpha) \cdot \sum_{i \in \mathcal{R}} RU_i$
CPU Frequency Scaling (CFS) (Wang et al., 2022a)	Docker Container	Physical Server	RL	Q-Learning	Instructions per second (IPS), CPU usage, Measured core frequency, SLO preservation ratio (SP) for latency/throughput	Core Frequency	$\beta \cdot ((IPS_t - IPS_{t-1}) / IPS_t)_{\Delta freq>0} + (1-\beta) \cdot SP_t$

We believe that FLASH significantly improves the practicality of applying ML in production cloud systems.

2 BACKGROUND AND MOTIVATION

2.1 ML-Centric Cloud Platforms

Recent advances in ML/RL have driven a shift in cloud system design and management. Public cloud providers have strong incentives to replace traditional system management agents with ML agents and to build ML-centric cloud platforms (Bianchini et al., 2020). The need for learning-augmented management stems from the fact that heterogeneous and complex decision-making spans across modern system lifecycles as cloud platforms become tremendously complex (Bianchini et al., 2020; Liang et al., 2020; Karthikeyan et al., 2023). These decisions govern how systems handle applications to satisfy user requirements in a particular runtime environment or cloud infrastructure (as shown in Figure 1). Various efforts (Cortez et al., 2017; Liang et al., 2020; Bianchini et al., 2020; Karthikeyan et al., 2023; Qiu et al., 2023b) have shown significant benefits in formulating production cloud system management into ML/RL-based tasks. We use three example ML agents (summarized in Table 1) to make our discussion concrete:

- **Resource configuration search (RCS)** is usually modeled as a regression task (Venkataraman et al., 2016; Yadwadkar et al., 2017; Klimovic et al., 2018; Eismann et al., 2021a). A supervised-learning-based predictor is trained to predict the application performance given the resource configurations, resource utilization, and other metrics. For example, Sizeless (Eismann et al., 2021a) leverages a fully connected neural network to predict the average execution times of a serverless function given a target function memory size based on the monitoring data when running the function with a base memory size[1].
- **Workload autoscaling (WA)** is modeled as a sequential decision-making problem to scale horizontally and/or vertically the controlled workload (e.g., the number of replicas and the size of each replica/pod for a Kubernetes Deployment) (Qiu et al., 2020; Wang et al., 2022b). RL is well suited for learning such policies, as it provides a tight feedback loop to explore the state action space and generate optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules) (Mao et al., 2016; Qiu et al., 2022). For example, FIRM (Qiu et al., 2020) leverages an RL model DDPG to adjust resources vertically (e.g., CPU/memory allocation) and scale horizontally (i.e., number of replicas). RL agents' goal is to maximize resource utilization while maintaining application SLOs.
- **CPU frequency scaling (CFS)** requires the agent to balance the workload performance improvements with the extra power cost when increasing the core frequency (Islam & Lin, 2017). SmartOverclock (Wang et al., 2022a) leverages an RL model, Q-Learning, to decide when and how much to scale the CPU core frequency.

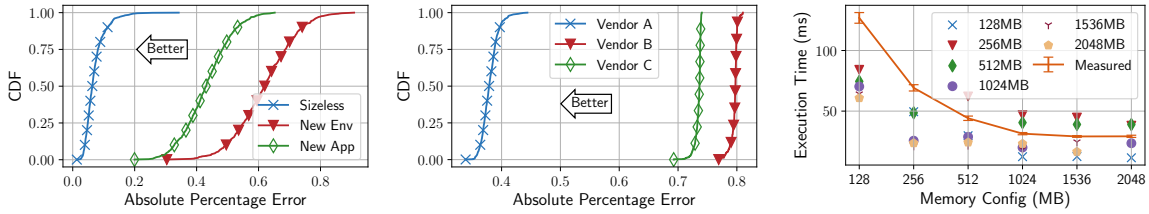
[1]In Sizeless, only the memory size is used because the CPU allocation is proportional to memory allocation on AWS Lambda.

2.2 Motivation: Why Existing Models Fail to Adapt?

Similar to traditional non-ML-based agents that require repeated profiling and parameter tuning, the trained models in ML-based agents require costly retraining to adapt to new applications or environments in heterogeneous and dynamically evolving (or even multi-cloud) datacenters (Mars & Tang, 2013; Hazelwood et al., 2018; Sriraman & Dhanotia, 2020; Stoica & Shenker, 2021; Qiu et al., 2023c):

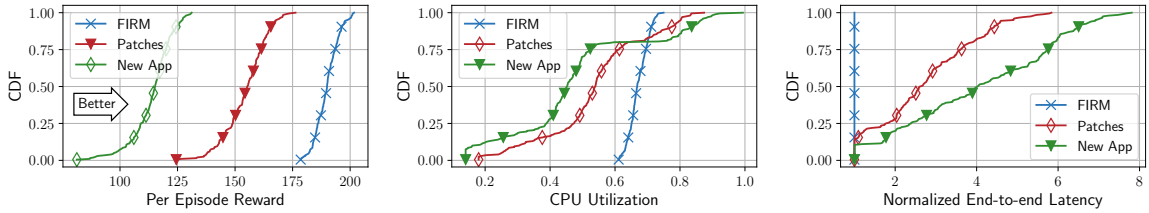
- *Environment diversity and infrastructure dynamics.* Heterogeneity exists in various types of datacenter infrastructure (Mars & Tang, 2013; Hazelwood et al., 2018). Compute or storage hardware upgrades can potentially alter the behavior of the existing system (Patterson, 2008; Liang et al., 2020). Network updates affect many factors, such as available link capacities, packet loss rates, and delay (Zheng et al., 2014; Singh et al., 2015).
- *Application diversity and workload dynamics.* Prior work has shown that there can be non-trivial differences among cloud applications (Kanev et al., 2015; Sriraman & Dhanotia, 2020; Wang et al., 2022b). In addition, cloud applications have increasingly adopted tighter and more frequent software update cycles (Neamtiu & Dumitraş, 2011; Gan et al., 2019), including changes in architectures, bug fixes or patches, and even payload data.

Such application/environment heterogeneity or dynamics could invalidate model assumptions (e.g., the optimal autoscaling policy changes with resource sensitivity) and cause the trained model to be suboptimal. Re-training models can be costly and requires a large amount of training data (Liang et al., 2020; Banerjee et al., 2020). To characterize such



(a) OpenWhisk dataset. *(b)* CloudBandit dataset. *(c)* An example of estimation error.

Figure 3. Testing trained Sizeless model on datasets collected from unseen cloud platforms or new applications.



(a) Reward degradation. *(b)* CPU utilization changes. *(c)* SLO preservation degradation.

Figure 4. Testing trained FIRM model on unseen applications or application updates.

adaptation requirements, we take the open-source implementation of Sizeless (for *RCS*) and FIRM (for *WA*) models, train them in the original setup described in their papers, and study the model performance degradation in new setups. Detailed descriptions of the model architecture, training, and datasets are deferred to appendices A.2 and A.3.

In RCS, we quantify the model performance degradation and show in Figure 3(a) and (b) the CDFs of the absolute percentage error (APE). The baseline (labeled as "Sizeless") is the CDF of the Sizeless model tested using the original Sizeless dataset, which has an average APE of 0.04 (consistent with the original paper (Eismann et al., 2021a)). When testing the trained model on unseen applications, the CDF (labeled as "New App") shows a 7.2× increase in the median APE. When testing the trained model (using data from cloud vendor A, identity hidden) on the same applications but running on vendor B/C, the CDFs show a 1.9–2.1× increase in median APE. Figure 3(c) shows an example of testing on an unseen application (i.e., Airline Booking). With different base memory configurations, *the prediction can be under-/over-estimated, leading to either application performance degradation or unnecessarily higher deployment costs*.

For WA, Figure 4(a) shows the per-episode reward degradation of FIRM's RL agents when encountering new applications (36.8% lower in median reward) or application updates (15.8% lower in median reward). We then further investigate the degradation regarding container CPU utilization and the 99th percentile application end-to-end latency. Figure 4(b) and (c) show that the RL reward degradation comes from both (1) over-allocation which leads to low utilization (e.g., median utilization when serving new applications is 39% compared to the FIRM baseline's 64%) and (2) under-allocation which leads to SLO violations (e.g., more than 25% agents have at least 5.8× higher 99th percentile end-to-end latency than the FIRM baseline). The degradation is primarily due to application workloads' heterogeneous behavior regarding resource utilization/demands and varying performance sensitivity to resource allocations (Qiu et al., 2020). *The learned mapping (i.e., the RL policy network) between RL states and the optimal actions is no longer valid and thus requires retraining.*

In addition to RCS and WA, we also find that fast model adaptation is required in other system management tasks (e.g., CFS) as shown in Appendix A.5.

2.3 Meta-learning Background

Meta-learning has been demonstrated (in fields like robotics control) to adapt well or generalize to new, previously unseen samples during training (Mishra et al., 2018). In the meta-learning training stage, rather than training the learner on a single environment to generalize to unseen "intra-environment" samples from a similar data distribution, a meta learner is trained on different distributions of environments, with the goal of learning a strategy that generalizes to unseen environments (i.e., "inter-environment"). Note that even though the new environment has never been encountered, adaptation is still possible when the new environment shares patterns similar to the ones encountered (Mishra et al., 2018; Hospedales et al., 2022).

Why Meta-learning? First, compared to meta-learning, training on a narrow distribution of (A_i, E_j) pairs results in poor generalization while simply training on a wide range of (A_i, E_j) pairs leads to average or suboptimal performance (Qiu et al., 2020; Xia et al., 2022). For example, resource autoscaling policies vary with resource consumption characteristics, workload sensitivity to different resource allocations, and heterogeneous service-level objectives. An alternative technique is called curriculum learning

(CL) (Narvekar et al., 2020; Xia et al., 2022). In our context, the ML agent can be trained with a sequence of (A_i, E_j) pairs ordered in terms of "difficulty" (which can be challenging to define and quantify). However, the goal of CL is to optimize the performance in the final task of the whole learning sequence (Narvekar et al., 2020). Meta-learning, in contrast, is to optimize for fast adaptability to a new task within a small number of model update steps.

Second, training one model per (A_i, E_j) pair entails significant training cost and model management overhead (Vartak & Madden, 2018; Sun et al., 2020), while there is a lack of a principled and systematic approach for clustering (A_i, E_j) pairs into groups and training one model per group. Even with transfer learning (Qiu et al., 2020), non-trivial retraining is still needed to adapt to a new (A_i, E_j). Meta-learning enables each ML agent to adapt to new (A_i, E_j) pairs in a systematic manner by identifying similar pairs through the learned representations of each (A_i, E_j) pair and leveraging shared knowledge and patterns from those similar pairs.

3 FLASH DESIGN

3.1 Overview and Architecture

Instead of developing and training one model per (A_i, E_j) pair, we design FLASH as a general framework that assists developers in developing ML agents with fast adaptability to new applications or deployment environments. Specifically, we focus on supervised learning (SL) and reinforcement learning (RL) because of their wide adoption in cloud system management tasks (Maas, 2020). FLASH's API (§3.4) guides agent developers through the agent-specific logic needed to facilitate fast model adaptation while remaining highly extensible to different use cases (as we show in §4). Figure 5 presents an architecture overview of FLASH.

Base Learner. FLASH models the ML agent (i.e., SL/RL agent) as a *base learner* with no model design changes.

- An SL model is trained with labeled data samples, meaning that each data point contains a *feature* vector and an associated *label*. The goal is to learn a function that maps from feature vectors to labels (prediction) with minimal error based on example input-output pairs in the training dataset. As an example, in Sizeless (Eismann et al., 2021a), an SL agent gets the features (e.g., resource consumption and execution time under a certain function memory size) for a serverless function and outputs the predicted function execution time for a target function memory size.
- In RL, an *agent* learns an optimal *policy* in a task modeled as a discrete-time Markov decision process (MDP). At time step t, the agent perceives a *state* $s_t \in S$ 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 a new state s_{t+1}. The entire sequence of transitions $\{(s_t, a_t, r_t)\}_{0 \le t \le T}$ is called a *trajectory* or *episode* of length T. The agent's goal is to learn

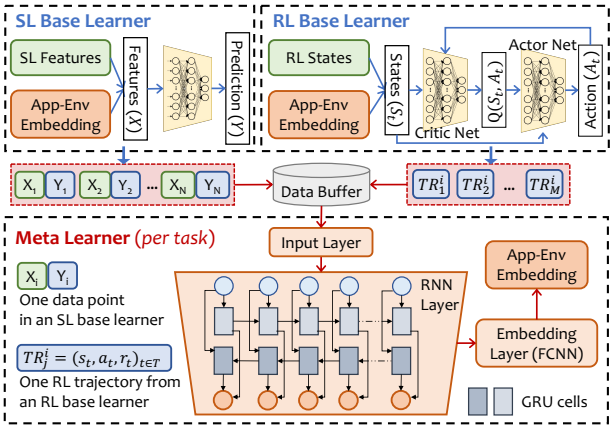


Figure 5. FLASH's architecture and integration between the meta learner and base learners (*either* with SL *or* RL).

a *policy* π_θ 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. As an example, an RL agent for workload autoscaling (Qiu et al., 2020) takes states (i.e., system metrics and application performance measurements) from Kubernetes and takes actions to determine the optimal scaling action.

Data Buffer. FLASH collects training and inference data from the SL or RL agents and stores the data *selectively* in a database. The selection criteria are based on SL and RL logic as described in §3.2. In SL, the data buffer consists of labeled data samples with both feature vectors and the label. In RL, the data buffer consists of a set of trajectories, each of which records the (s_t, a_t, r_t) transition sequences when the RL agent interacts with the system.

Meta Learner. To rapidly adapt to a new/updated (A_i, E_j) pair in a system management task, the *meta learner* selects SL samples or RL trajectories from the data buffer and generates an *embedding* that accurately represents this (A_i, E_j) pair. The embedding is then fed to the base learner (i.e., the SL or RL agent) as part of its feature vector or state vector. The base learner leverages the embedding to adapt (fine-tune) its model or policy by differentiating heterogeneous applications and environments. See §3.2 and §3.3 for more details about the design, training, and inference.

3.2 Meta Learner Design

We introduce *embedding-based meta-learning* that is designed to explicitly model the *individuality* of each (A_i, E_j) pair in a system management task. Instead of meta-learning the architectural or algorithmic level configurations (e.g., parameter initialization, learning rate, or neural network architecture), FLASH's meta learner learns to generate an *embedding* that projects the application- and environment-specific characteristics to a vector space. On this projected vector space, (A_i, E_j) pairs with similar characteristics are projected to neighboring locations, while those different

ones are projected to locations far from each other.

As shown in Figure 5, the meta learner (**per task** and thus **per SL/RL base learner**)'s network architecture consists of (1) an input layer, (2) a recurrent neural network (RNN) layer, and (3) a fully connected neural network (FCNN) layer (i.e., the embedding layer).

Input Layer. The input layer selects what kind of information the meta learner retrieves from the data buffer for embedding generation. Based on the insight that the training datasets of the developed ML models already contain spatial-temporal characteristics of each (A_i, E_j) pair, labeled data samples (in SL) and trajectories (in RL) are used as inputs to the meta learner. However, for RL, simply using all trajectories is computationally intensive in practice. Instead, we choose $M/2$ trajectories with the highest rewards and $M/2$ trajectories with the lowest rewards, excluding lower-reward trajectories generated during the initial training stage due to their lack of representativeness.

RNN Layer. We use a bidirectional GRU (a special class of RNNs) (Schuster & Paliwal, 1997; Sutskever et al., 2011) 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 hidden states for a long time. Hence, they are effective in capturing both spatial and temporal patterns. In addition, a unidirectional RNN has the limitation that it processes inputs in strict temporal order, so the current input has the context of previous inputs only (not the future). Bidirectional RNNs, on the other hand, duplicate the processing chain so that the inputs are processed in both forward and backward orders to include future contexts as well. It should be noted that we do not use the memory augmentation technique (Santoro et al., 2016) for our RNN-based meta learner because we find that the feature space in system management tasks is not as high-dimensional as those of computer vision tasks, and the RNN hidden states suffice to provide good representations.

Embedding (FCNN) layer. The output from the RNN layer of the meta learner is fed to a fully connected two-layer neural network to generate an embedding (i.e., a fixed-size vector) that is used to fingerprint or represent the (A_i, E_j) pair with which the base learner is dealing. As shown in Figure 5, the generated embedding is finally concatenated by the base learner as part of the SL feature vector or RL state vector at each time step.

3.3 Pre-training and Fast Adaptation

FLASH is pre-trained on a pool of (A_i, E_j) pairs (i.e., pre-training pool) and fine-tuned to novel (A_i, E_j) pairs in the adaptation pool, as shown in Figure 6.

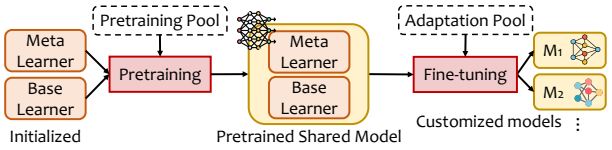


Figure 6. FLASH's pre-training and adaptation workflow.

Pretraining. FLASH is exposed to a pretraining pool of (A_i, E_j) pairs and trained to discriminate the *individuality* of each pair with meta-learning. The loss value generated from the base learner is backward-propagated to update the model parameters of both the base learner and the meta learner. FLASH uses an ordinary gradient descent update of RNN-based networks (inspired by SNAIL (Mishra et al., 2018)) with a hidden state reset at a switch of (A_i, E_j) pairs. After convergence, the model checkpoint of the meta learner and the base learner (which is called the *pretrained shared model*) is then served as a common basis to fine-tune customized models for different (A_i, E_j) pairs in the adaptation stage. In addition to individuality, FLASH also captures the *commonality* across (A_i, E_j) through the shared model (or shared policy in RL) that is conditioned on the embeddings, which allows a base learner to adapt to different pairs with the same shared model parameters. *The resultant shared model in* FLASH *is analogous to a pre-trained "foundation model" for a given cloud system management task.*

Adaptation. Since the pretrained shared model is conditioned on application-/environment-specific embeddings, the adaptation process only requires limited exposure (i.e., a few SL samples or RL trajectories to feed to the meta learner) for embedding generation. Note that even though the new (A_i, E_j) pair has never been encountered, adaptation is still possible when the new pair shares similar patterns with the encountered ones (Mishra et al., 2018; Hospedales et al., 2022). For the pairs with quite dissimilar patterns (identifiable based on the distance in the embedding space as we describe in §5.5), the shared model/policy conditioned on the embedding can be further fine-tuned to adapt to the optimal customized model/policy. The model parameters of the meta learner are **fixed**. *Overall, the pretrain-finetune paradigm provides an efficient way to balance pre-training cost with the need for fast adaptation for heterogeneous cloud applications and environments.*

Interpreting Embeddings from a Systems Perspective. Figure 7 visualizes the key idea of embedding in the workload autoscaling task. From a systems perspective, both the *spatial* and *temporal* characteristics of the applications are encoded and mapped onto a low-dimensional latent vector space. Applications with similar characteristics are projected to locations that are close to each other on that vector space. By calculating the *cosine similarity* and the *Euclidean distance* between any two generated embedding vectors, we can get a two-dimensional similarity measurement. In Figure 7 (left), the sensitivity of application performance

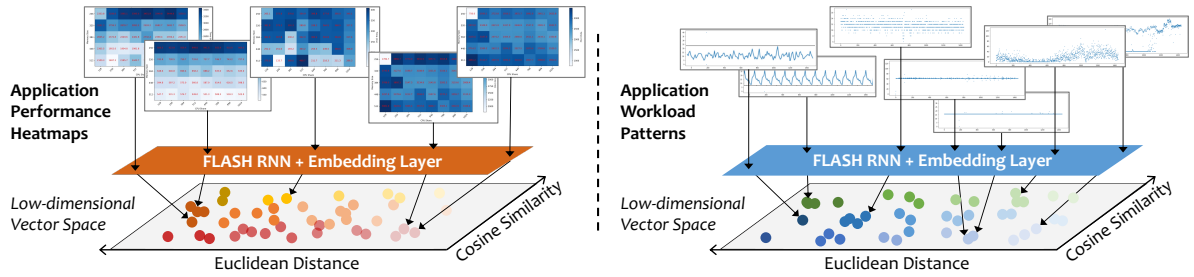


Figure 7. An example illustrating how *embedding* encodes spatial and temporal characteristics from a systems perspective.

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 Figure 7 (right), the application load-per-second time series can represent the temporal characteristics. Applications with similar spatial-temporal patterns are projected to adjacent locations in the output vector space. We show in §5.5 how the defined similarity metric can be used for adaptation predictability.

3.4 FLASH Programming Interface

FLASH exposes a unified API to ML agent developers that supports both SL and RL as shown in Listing 1. Each interface has four operations that the developers need to implement for integration with FLASH.

(1) Register the SL or RL base learner by providing the name of the system management task (there will be one meta learner per task) and the input shapes. The meta learner then starts to initialize the meta-learning network by configuring the dimensions. The input dimension is `numSamples*(F.shape+P.shape)` for SL and `numTrajs*(S.shape+A.shape+1)` for RL.
(2) Insert the data samples (for SL) or trajectory samples (for RL) into the data buffer. The meta learner will update the data buffer based on the selection criteria in §3.2.
(3) Get the current embedding. The base learner will take the embedding generated by the meta learner and concatenate it to the feature vector in SL or the state vector in RL.
(4) Turning on and off the meta learner model parameter update. This function transitions between the meta-learning training and inference stages. When turned off, the shared model (i.e., pre-trained meta learner and base learner) parameters will not be updated. Practically, developers can retrain the meta learner every week or so to incrementally minimize out-of-distribution probabilities based on the frequency that new (A_i, E_j) pairs are added to the task.

4 FLASH Use Cases

We study three use cases of FLASH, each of which corresponds to a different system management task. The three chosen ML agents differ in input data, ML models they use, and the output they produce (i.e., prediction or action).

```
// For supervised learning agents
interface SLModel<F: Feature, P: Label> {
    void RegisterBaseLearner(string, F, P);
    void InsertDataSamples(List[<F, P>]);
    Embedding<E> GetEmbedding();
    void ToggleMetaModelUpdate(bool);
}
// For reinforcement learning agents
interface RLModel<S:State, A:Action, R:
     Reward> {
    void RegisterBaseLearner(string, S, A);
    void InsertTrajSamples(List[<S, A, R>]);
    Embedding<E> GetEmbedding();
    void ToggleMetaModelUpdate(bool);
}
```

Listing 1. FLASH APIs for SL and RL agents. `SLModel` *is parameterized by the feature and label type of the data samples.* `RLModel` *is parameterized by the state, action, and reward type of the RL trajectories.*

Resource Configuration Search. In this case study, we focus on the most recent work, Sizeless (Eismann et al., 2021a), which applied supervised learning in predicting the optimal memory size of serverless functions based on monitoring data for a single memory size. As mentioned in §2.2, the diversity of application workloads and compute infrastructure requires model adaptation. We take the open-source implementation of the Sizeless model as the base learner and integrate it with FLASH meta learner. The resultant agent is referred to as **FLASH-Sizeless**.

In Sizeless, the task of predicting the execution time of a serverless function is formulated as a regression problem. The input includes a target memory size and monitoring data (e.g., heap usage) when running at base memory size(s), and the output is the estimated execution time (see Table 1). The users can then decide which memory configuration to choose, given the cost and estimated performance when running with that configuration. Sizeless trains a fully connected neural network with ReLU as the activation function based on the average execution time and monitored resource consumption metrics when running with the base memory size(s). More details are shown in Appendix A.2. The features and labels are recorded to the data buffer by calling `InsertDataSamples()`. The embedding is retrieved with `GetEmbedding()` and then appended to the original

feature vector used in the fully connected neural network.

Workload Autoscaling. We integrate FLASH with an open-sourced state-of-the-art RL-based autoscaler from FIRM (Qiu et al., 2020). The resultant agent is referred to as **FLASH-FIRM**. FIRM uses an actor-critic RL algorithm called DDPG (Lillicrap et al., 2016). 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 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, $SP = 1$; otherwise, $0 < SP < 1$. The reward function is then defined the same as in FIRM (Qiu et al., 2020), $r_t = \alpha \cdot SP_t \cdot |\mathcal{R}| + (1 - \alpha) \cdot \sum_{i \in \mathcal{R}} RU_i$, where $\mathcal{R}$ is the set of resources. The RL algorithm is trained in an episodic setting. In each episode, the agent manages the autoscaling of the application for a fixed number of RL time steps. The resulting state-action-reward series is called a trajectory. More details are shown in Appendix A.3.

FLASH receives an RL trajectory when the base learner calls `InsertTrajSamples()` after each episode. To handle variable-length trajectories, each trajectory in the buffer is padded with 0s to have equal lengths. Then, the state, action, and reward tensors along the last dimension are concatenated to create the input for the RNN layer. The same trajectory padding is also done for the last case study (CFS). The embedding is retrieved by calling `GetEmbedding()` and then appended back to the RL state vector.

CPU Frequency Scaling. Our last agent, SmartOverclock, is an intelligent on-node overclocking agent proposed by Microsoft (Wang et al., 2022a). We adopt SmartOverclock as the base learner to integrate with FLASH, and the resultant agent is referred to as **FLASH-SmartOverclock**. CPU frequency scaling presents opportunities for substantial performance improvements or saving of CPU cores on different sets of workloads (Chen & Marculescu, 2015; Jalili et al., 2021). Despite the benefits of overclocking, it significantly increases node power consumption and can shorten processor lifetimes. SmartOverclock learns to balance application workload performance improvements with extra power cost by using RL to decide when and how much to scale the CPU core frequency.

SmartOverclock uses an RL model, Q-Learning (Baseline3, 2023). At each time step t (every 1 second), the agent monitors the average Instructions Per Second (IPS) performance counter across the cores of each VM and learns when to adjust the core frequency. Table 1 shows the RL model's state and action spaces. Since the goal is to increase the frequency only when the workload benefits from it, e.g., (a) higher CPU frequencies increase the IPS, or (b) the SLO preservation ratio (SP_t) is high, the reward function is defined as $r_t = \beta \cdot ((IPS_t - IPS_{t-1})/IPS_t)_{\Delta freq>0} + (1-\beta) \cdot SP_t$. More details of the model and this case study are deferred to Appendix A.4. Similarly to the base learner implemented for FIRM, the base learner calls `InsertTrajSamples()` after each RL episode to save trajectories and retrieves the embedding by calling `GetEmbedding()`. The embedding is then appended to the state vector taken by the Q-Network (in Q-Learning) to complete the value function.

5 EVALUATION

In each case study, we evaluate (a) the performance degradation of the ML agent with FLASH when testing for new applications/environments, and (b) the efficacy of FLASH's meta learner in fast model adaptation (in §5.1–§5.3). We then evaluate the overhead of FLASH (§5.4) in both training and inference. In addition, we show that a *similarity metric* derived from the generated embeddings can be leveraged to enhance the predictability of adaptation cost (§5.5).

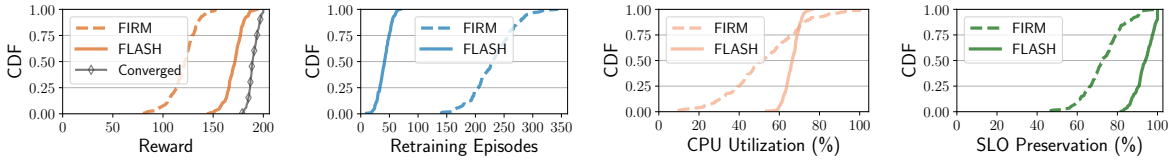
5.1 FLASH-Sizeless

Setup. Three datasets are used for evaluation: (a) the original Sizeless dataset (Eismann et al., 2021a) consisting of 2000 serverless applications, (b) the OpenWhisk dataset, which we collected on a local OpenWhisk cluster following the same methodology as Sizeless but with CPU allocation added as the resource configuration in addition to function memory size, and (c) the CloudBandit dataset (Lazuka et al., 2022) for 30 VM-based applications which covers resource configuration (e.g., VM type and vCPU count), performance metrics, system metrics on three public cloud platforms. We run ten iterations of five-fold cross-validation with a 50/50 training/testing split for each dataset. Datasets (a) and (b) are used to evaluate the model's robustness across different applications, while dataset (c) is used to evaluate the robustness across different cloud infrastructures (i.e., environments). Details of the dataset are deferred to Appendix A.2.

Prediction Error without Adaptation. Table 2 shows the model performance comparison where the evaluation metric is mean absolute percentage error (MAPE). We evaluate the model performance on unseen applications (for Sizeless and OpenWhisk datasets) and environments (for CloudBandit dataset) with X-shot ($X \in [1, 2, 3]$) settings, where X is the number of data samples Sizeless agent uses as the base configuration(s) to predict the application performance under the target configuration. The Sizeless dataset leads to the lowest testing MAPE (on average 0.37 for the vanilla Sizeless model and 0.04 for FLASH-Sizeless) because function memory size is the only configuration being considered. The OpenWhisk dataset contains CPU allocation in addition to function memory size, and thus the average testing MAPE is higher (0.64 for the Sizeless model and 0.3 for FLASH-Sizeless). The CloudBandit dataset results in the

Table 2. Sizeless agent prediction error (i.e., MAPE) with and without FLASH. *The number of samples used as the base configuration in the task of resource configuration search is indicated as X-shot as in few-shot learning.*

Dataset	**Sizeless**			**OpenWhisk**			**CloudBandit**		
# of Samples	1-shot	2-shot	3-shot	1-shot	2-shot	3-shot	1-shot	2-shot	3-shot
Sizeless (training)	0.040	0.036	0.035	0.316	0.258	0.236	0.610	0.439	0.424
Sizeless (testing)	0.360	0.400	0.336	0.823	0.552	0.540	0.985	0.889	0.798
FLASH-Sizeless (training)	0.038	0.036	0.032	0.321	0.247	0.259	0.624	0.416	0.435
FLASH-Sizeless (testing)	0.046	0.038	0.034	0.357	0.263	0.275	0.649	0.424	0.497
Improved (testing)	87.22%	**90.50%**	89.88%	**56.62%**	52.36%	49.07%	34.11%	**52.31%**	37.72%



(a) Reward Degradation. *(b)* Retraining Cost. *(c)* Resource Utilization. *(d)* Performance.

Figure 8. Comparison of the performance and retraining cost of the FIRM model with FLASH.

highest testing MAPE because the collected data is across both heterogeneous workloads and three different cloud providers. Lastly, the testing MAPE drops when the number of samples used as the base configurations for prediction increases from 1-shot to 3-shot since the agent can leverage more runtime information. FLASH-Sizeless achieves up to 90.5%, 56.6%, and 52.31% lower MAPE compared to the original Sizeless model in datasets Sizeless, OpenWhisk, and CloudBandit, respectively. The improvement on the CloudBandit dataset is the lowest because both the Sizeless and OpenWhisk datasets contain detailed system metrics (e.g., user/system CPU time, heap used, and cache misses), which helps generate more expressive embeddings.

5.2 FLASH-FIRM

Setup. We leverage the open-source application generators and representative serverless benchmarks from Sizeless to generate 1000 synthetic applications (as described in Appendix A.3) because serverless workloads are highly dynamic (and thus require autoscaling) and rely on the provider to manage the resources. For RL agent training and inference, we use real-world datacenter traces (Zhang et al., 2021b) released by Microsoft Azure, collected over two weeks in 2021. Next, we deploy the selected workloads as Deployments in a five-node Kubernetes cluster in a public cloud and ran an RL-based multi-dimensional autoscaler with each Deployment, controlling both the number of replicas (horizontal scaling) and the container sizes (vertical scaling). All nodes run Ubuntu 18.04 with four cores, 16 GB memory, and a 200 GB disk.

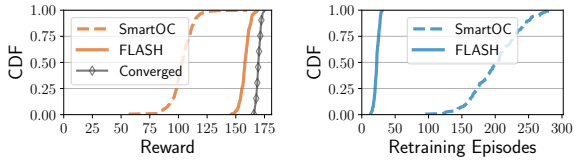
In the evaluation, we repeat five experiment runs wherein each run, the pretraining of FLASH-FIRM is done on 200 randomly selected applications (i.e., the pretraining pool in Figure 6), and the adaptation evaluation is done on the remaining 800 applications (i.e., the adaptation pool). A sensitivity study on the pretraining pool sizes is deferred to Appendix A.7.

Reward Drop without Adaptation. To evaluate the performance degradation without retraining, we design an autoscaler A/B test where FLASH-FIRM agent and the original FIRM agent are the two variants controlling the autoscaling for the same set of traces. We repeat the A/B test 100 times. In each test, we randomly select an application from the adaptation pool and train the agent until convergence. We then randomly select ten other different applications from the pool for reward drop evaluation (i.e., in RL policy-serving). Figure 8(a) shows the CDF of the per-episode reward, and we find that FLASH-FIRM reduces the average reward drop percentage from the baseline (i.e., the agent trained to convergence on the testing application), labeled as "Converged" in Figure 8(a), from 37% to 10.5%.

Adaptation Cost. FIRM (Qiu et al., 2020) leverages transfer learning (TL) to retrain an RL agent for a new application based on previous RL experience gained for known applications. In the TL-based approach, the model parameters (weights) are shared between the agents managing the known workload and the new workload. We measure the retraining time, CPU utilization, and SLO preservation of FIRM and FLASH. Results are shown in Figure 8(b) – (c). We find that, on average, FLASH-FIRM adapts 5.5× faster than FIRM, resulting in 77.6% less performance degradation and 5.1× less CPU utilization deficit (i.e., the utilization gap from the converged RL policy) during adaptation.

5.3 FLASH-SmartOverclock

Setup. The application workloads in training and testing for this case study are the same as in evaluating FLASH-FIRM (see §5.2) and are deployed in Docker containers on three different types of processors: Intel Xeon E5-2683 v3, Intel Xeon CPU E5-2695 v4, and AMD EPYC 7302P. Therefore, the cross-application/processor settings require model adaptation across both applications and environments.



(a) Reward Degradation. *(b)* Retraining Cost.

Figure 9. Comparison of the performance and retraining cost of the SmartOverclock model with FLASH.

Reward Drop without Adaptation. Similar to §5.2, we design and repeat the A/B test 100 times, comparing FLASH-SmartOverclock with the vanilla SmartOverclock agent. In each test, we randomly select a processor type and an application from the application pool and then train the agent until convergence. We then randomly select ten other different applications from the pool, each running on a randomly selected processor, for reward drop evaluation. Figure 9(a) shows the CDF of the per-episode reward, and we find that FLASH-SmartOverclock improves the average reward drop percentage from the baseline (i.e., labeled as "Converged" in the figure) from 39% to 7.1%.

Adaptation Cost. Similar to §5.2, we evaluate the retraining cost by comparing FLASH with TL regarding the number of RL episodes needed to converge. Figure 9(b) shows the CDF of the RL retraining cost and, on average, FLASH-SmartOverclock adapts 9.2× faster than TL.

5.4 FLASH Overhead

We evaluate the training and inference overhead of FLASH introduced to each ML agent (as shown in Table 3). At the inference stage, generating embeddings and including the embedding in the forward pass process of the original neural network introduce additional latency. The inference overhead can be up to 3.9× (i.e., from 1.5 ms to 5.9 ms for SmartOverclock). However, compared to the second (s)-level RL time steps, such sub-10 ms overhead did not affect the RL training convergence or policy-serving performance. At the training stage (including the meta learner), the model update latency overhead is up to 4.1× (from 1.21 s to 4.95 s for FIRM), leading to a pre-training time of ∼5.2 hours with an NVIDIA Tesla V100 (16 GB) GPU. Model update of the RNN/GRU layer accounts for 75% of the total model update latency. Therefore, after meta learner training and meta learner model parameters are fixed, the model update overhead of the base learner can be negligible compared to the model update latency of the original ML/RL agent (a 3% increase, i.e., 1.25 s compared to 1.21 s).

Table 3. Training and inference overhead of FLASH. *Inference overhead (ms) includes the latency of generating the embedding, and model update overhead (s) includes the latency of updating the meta learner during pretraining.*

	Model Inference Latency (ms)			Model Update Latency (s)		
	Sizeless	FIRM	SmartOC	Sizeless	FIRM	SmartOC
No FLASH	0.4 ± 0.1	2.3 ± 0.6	1.5 ± 0.3	5.4 ± 1.9	1.21 ± 0.4	0.88 ± 0.2
With FLASH	1.4 ± 0.3	8.1 ± 3.0	5.9 ± 1.7	16.8 ± 3.2	4.95 ± 0.8	3.32 ± 0.9

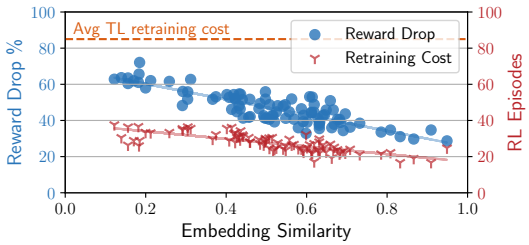


Figure 10. Correlation between the embedding similarity and (a) the reward drop percentage without retraining (in blue) and (b) the retaining cost with FLASH (in red).

5.5 FLASH Adaptation Predictability

The embedding generated by the meta learner not only enables fast model adaptation to novel cloud applications or environments but also provides predictability regarding both (a) performance drop and (b) retraining overhead. Given two embeddings e_i and e_j, we found that the embedding similarity defined as $S(e_i, e_j) = (1 - ED(e_i, e_j) + CS(e_i, e_j))/2$ can be a good indicator, where $ED(e_i, e_j)$ is the normalized Euclidean distance and $CS(e_i, e_j)$ is the Cosine similarity between the two embeddings. Figure 10 illustrates the correlation between embedding similarity and (a) the reward drop percentage (in blue) and (b) the number of RL episodes needed for retraining (in red) with FLASH in the example task of workload autoscaling. Performance and retraining cost prediction help understand the cost before deploying ML agents for new applications or to new environments.

6 RELATED WORK

Meta-learning for Systems. ResTune (Zhang et al., 2021a) leverages meta-learning to optimize hyperparameters that boost Bayesian optimization on database performance tuning knobs. Xue et al. (2022) employ neural processes, a meta-learning model, to train a fully connected neural network to predict workload CPU utilization. However, neural processes cannot adapt the model parameters on the fly during inference based on the current task or context (Rusu et al., 2019). Other examples of meta-learning model parameter initialization include PSO (Leka et al., 2023) and DMRO (Qu et al., 2021), which do not explicitly and adequately model the individuality of tasks as FLASH does through its interpretable embedding generation.

Zero/Few-shot Learning for Systems. Hilprecht & Binnig (2021) first propose the idea of zero-shot learning for databases (e.g., query cost estimation and index selection). By encoding database queries with transferable features, an ML-driven model can generalize across databases since feature representations remain consistent. Zero/few-shot learning has also been applied in intrusion attack detection

in networks (Zhang et al., 2020) and microservices (Liang et al., 2022). Meta-learning can be complementary by learning a good representation that can generalize well to help with zero/few-shot learning (Verma et al., 2020).

Curriculum Learning for RL. We consider curriculum learning (Narvekar et al., 2020) (e.g., Genet (Xia et al., 2022)) in the networking domain) as an orthogonal technique for meta-learning. In our context, the ML agent can be trained with a sequence of ⟨`application`, `environment`⟩ pairs ordered in terms of "difficulty". While curriculum learning aims to optimize the asymptotic performance in the final task of the learning sequence (Narvekar et al., 2020), meta-learning provides theoretically proved generalizability across tasks.

7 Discussion and Limitations

Meta Learner Model Size/Complexity. Our experiments show that a two-layer bidirectional GRU (RNN) followed by a fully connected layer already provides 5.5× faster model adaptation compared to transfer learning (in the task of workload autoscaling). We plan to conduct larger-scale experiments to investigate the necessity of an extreme-size model or a more complex model architecture (e.g., Transformers (Vaswani et al., 2017)). However, a larger model may lead to unnecessarily higher pre-training costs and inference overhead, which could be detrimental to latency-sensitive online system management tasks (e.g., job scheduling (Mao et al., 2019b) or autoscaling (Qiu et al., 2021)).

Feasibility. There have been rich monitoring or profiling data in modern datacenters that enables pre-training across (A_i, E_j) pairs with meta-learning. For instance, Google-Wide Profiling (GWP) (Ren et al., 2010; Kanev et al., 2015) and Monarch (Adams et al., 2020) are profiling infrastructures for datacenters, providing performance insights for machines and cloud applications. Cluster managers such as Borg (Verma et al., 2015) monitor a full range of applications and generate task-event (e.g., kill and pending) and resource usage monitoring (e.g., CPU usage, memory usage, and disk I/O time) that provide rich characteristics about running application workloads (Kanev et al., 2015).

Amortization of Pretraining Overhead. Pretraining across a distribution of (A_i, E_j) pairs can require a large amount of training overhead (e.g., pretraining on 200 pairs costs 5.2 hours as mentioned in §5.4). However, adapting for potentially *larger-scale* novel (A_i, E_j) pairs requires substantially fewer model update iterations. This trade-off allows us to reduce the per-pair training cost (i.e., *amortization*, as shown in Appendix A.7), especially for diverse or constantly evolving (A_i, E_j) pairs in cloud environments.

Meta Learner Retraining. The base learner and meta learner abstractions in FLASH enable an ML-for-systems base learner model to be used with no changes to the base learner model design or training algorithms. Base learner inputs/outputs (see Table 1) are used as inputs to the meta learner. Therefore, substantial feature changes (adding new features or removing existing features) in the base learner can lead to retraining the meta learner from scratch.

Cross-Task Model Adaptation. FLASH enables fast model adaptation to new (A_i, E_j) pairs *within* each task so one meta learner is trained per task. Therefore, the learned embeddings are bound to a particular task. An exception could be that if applications are the same but simply the tasks are different (e.g., for the same containers, task A is to allocate the initial resource configuration, and task B is autoscaling), they might be able to share the same embeddings because the embeddings essentially represent the application features. Future work has to be done for general cross-task adaptation (Qiu et al., 2023a).

8 Conclusion

This paper explored the challenges of model adaptation toward ML-centric cloud platforms in practice. We presented FLASH, a general and extensible framework for developing ML agents that can rapidly adapt to new, previously unseen cloud applications or environments. To demonstrate FLASH, we implemented three agents and experimentally showed how FLASH improves model adaptation with meta-learning and a pretrain-finetune paradigm. FLASH is open-sourced at `https://gitlab.engr.illinois.edu/DEPEND/flash`.

Acknowledgments

We thank the anonymous reviewers for providing their valuable feedback. This work is supported by 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.

References

Adams, C., Alonso, L., Atkin, B., Banning, J., Bhola, S., Buskens, R., Chen, M., Chen, X., Chung, Y., Jia, Q., Sakharov, N., Talbot, G., Tart, A., and Taylor, N. Monarch: Google's planet-scale in-memory time series database. *Proceedings of the VLDB Endowment*, 13(12): 3181–3194, 8 2020. ISSN 2150-8097.

Banerjee, S., Jha, S., Kalbarczyk, Z. T., and Iyer, R. K. Inductive-bias-driven reinforcement learning for efficient scheduling in heterogeneous clusters. In *Proceedings of the 37th International Conference on Machine Learning*, pp. 629–641, Cambridge, MA, USA, 2020. PMLR.

Baseline3, S. Deep Q-Network (DQN) Documentation.

https://stable-baselines3.readthedocs.io/en/master/modules/dqn.html, 2023. Accessed: 2023-03-29.

Bianchini, R., Fontoura, M., Cortez, E., Bonde, A., Muzio, A., Constantin, A.-M., Moscibroda, T., Magalhaes, G., Bablani, G., and Russinovich, M. Toward ML-centric cloud platforms. *Communications of the ACM*, 63(2): 50–59, jan 2020. ISSN 0001-0782.

Chen, Z. and Marculescu, D. Distributed reinforcement learning for power limited many-core system performance optimization. In *Proceedings of the 2015 Design, Automation and Test in Europe Conference and Exhibition (DATE 2015)*, pp. 1521–1526, 2015.

Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource Central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017)*, pp. 153–167, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853.

Eismann, S., Bui, L., Grohmann, J., Abad, C., Herbst, N., and Kounev, S. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference (Middleware 2021)*, pp. 248–259. Association for Computing Machinery, 2021a. ISBN 9781450385343.

Eismann, S., Scheuner, J., van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C. L., and Iosup, A. Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39, 2021b. doi: 10.1109/MS.2020.3023302.

Gan, Y., Zhang, Y., Hu, K., Cheng, D., He, Y., Pancholi, M., and Delimitrou, C. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*, pp. 19–33, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405.

Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., Law, J., Lee, K., Lu, J., Noordhuis, P., Smelyanskiy, M., Xiong, L., and Wang, X. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*, pp. 620–629, 2018.

Hilprecht, B. and Binnig, C. One model to rule them all: Towards zero-shot learning for databases. *arXiv preprint arXiv:2105.00642*, 2021.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9(8):1735–1780, nov 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735.

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

Islam, F. M. M. u. and Lin, M. Hybrid DVFS scheduling for real-time systems based on reinforcement learning. *IEEE Systems Journal*, 11(2):931–940, 2017. doi: 10.1109/JSYST.2015.2446205.

Jalili, M., Manousakis, I., Goiri, I. n., Misra, P. A., Raniwala, A., Alissa, H., Ramakrishnan, B., Tuma, P., Belady, C., Fontoura, M., and Bianchini, R. Cost-efficient overclocking in immersion-cooled datacenters. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA 2021)*, pp. 623–636, 2021. doi: 10.1109/ISCA52012.2021.00055.

Jay, N., Rotman, N. H., Godfrey, P., Schapira, M., and Tamar, A. A deep reinforcement learning perspective on internet congestion control. In *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*. PMLR, 2019.

Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., and Brooks, D. Profiling a warehouse-scale computer. *SIGARCH Computer Architecture News*, 43(3S):158–169, jun 2015. ISSN 0163-5964.

Karthikeyan, A., Natarajan, N., Somashekar, G., Zhao, L., Bhagwan, R., Fonseca, R., Racheva, T., and Bansal, Y. SelfTune: Tuning cluster managers. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2023)*, pp. 1097–1114, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5.

Klimovic, A., Litz, H., and Kozyrakis, C. Selecta: Heterogeneous cloud storage configuration for data analytics. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (ATC 2018)*, pp. 759–773, USA, 2018. USENIX Association. ISBN 9781931971447.

Lazuka, M., Parnell, T., Anghel, A., and Pozidis, H. Search-based methods for multi-cloud configuration. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD 2022)*, pp. 438–448, 2022. doi: 10.1109/CLOUD55607.2022.00067.

Leka, H. L., Fengli, Z., Kenea, A. T., Hundera, N. W., Tohye, T. G., and Tegene, A. T. PSO-based ensemble meta-learning approach for cloud virtual machine resource usage prediction. *Symmetry*, 15(3):613, 2023.

Liang, C.-J. M., Xue, H., Yang, M., Zhou, L., Zhu, L., Li, Z. L., Wang, Z., Chen, Q., Zhang, Q., Liu, C., and Dai, W. AutoSys: The design and operation of learning-augmented systems. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (ATC 2020)*, USA, 2020. USENIX Association. ISBN 978-1-939133-14-4.

Liang, W., Hu, Y., Zhou, X., Pan, Y., and Wang, K. I.-K. Variational few-shot learning for microservice-oriented intrusion detection in distributed industrial IoT. *IEEE Transactions on Industrial Informatics*, 18(8):5087–5095, 2022. doi: 10.1109/TII.2021.3116085.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. In Bengio, Y. and LeCun, Y. (eds.), *Proceedings of the 4th International Conference on Learning Representations (ICLR 2016)*, 2016. https://arxiv.org/abs/1509.02971.

Ma, Y., Tian, H., Liao, X., Zhang, J., Wang, W., Chen, K., and Jin, X. Multi-objective congestion control. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys 2022)*, pp. 218–235, New York, NY, USA, 2022. Association for Computing Machinery.

Maas, M. A taxonomy of ML for systems problems. *IEEE Micro*, 40(5):8–16, 2020. doi: 10.1109/MM.2020.3012883.

Mao, H., Alizadeh, M., Menache, I., and Kandula, S. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNet 2016)*, pp. 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

Mao, H., Negi, P., Narayan, A., Wang, H., Yang, J., Wang, H., Marcus, R., Khani Shirkoohi, M., He, S., Nathan, V., et al. Park: An open platform for learning-augmented computer systems. *Advances in Neural Information Processing Systems (NeurIPS 2019)*, 32, 2019a. https://proceedings.neurips.cc/paper/2019.

Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., and Alizadeh, M. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM 2019)*, pp. 270–288, New York, NY, USA, 2019b. Association for Computing Machinery.

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

Mishra, N., Rohaninejad, M., Chen, X., and Abbeel, P. 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.

Narvekar, S., Peng, B., Leonetti, M., Sinapov, J., Taylor, M. E., and Stone, P. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research*, 21(1), jan 2020. ISSN 1532-4435.

Neamtiu, I. and Dumitraş, T. Cloud software upgrades: Challenges and opportunities. In *2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pp. 1–10. IEEE, 2011.

Patterson, D. A. Technical perspective: The data center is the computer. *Communications of the ACM*, 51:105, 2008.

PyTorch. Gated Recurrent Unit (GRU) Documentation. https://pytorch.org/docs/stable/generated/torch.nn.GRU.html, 2023. Accessed: 2023-03-29.

Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z. T., and Iyer, R. K. 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)*, pp. 805–825, Berkeley, CA, USA, November 2020. USENIX Association.

Qiu, H., Jha, S., Banerjee, S. S., Patke, A., Wang, C., Hubertus, F., Kalbarczyk, Z. T., and Iyer, R. K. Is function-as-a-service a good fit for latency-critical services? In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, pp. 1–8, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391726.

Qiu, H., Mao, W., Patke, A., Wang, C., Franke, H., Kalbarczyk, Z. T., Başar, T., and Iyer, R. K. Reinforcement learning for resource management in multi-tenant serverless platforms. In *Proceedings of the 2nd European Workshop on Machine Learning and Systems (EuroMLSys 2022)*, pp. 20–28, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392549. doi: 10.1145/3517207.3526971. URL https://doi.org/10.1145/3517207.3526971.

Qiu, H., Mao, W., Franke, C. W. H., Kalbarczyk, Z. T., Basar, T., and Iyer, R. K. On the promise and challenges of foundation models for learning-based cloud systems management. In *Workshop on Machine Learning for Systems at NeurIPS 2023*, 2023a.

Qiu, H., Mao, W., Wang, C., Franke, H., Youssef, A., Kalbarczyk, Z. T., Başar, T., and Iyer, R. K. AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 387–402, 2023b.

Qiu, H., Zhang, L., Franke, Chen, W., Hubertus, Kalbarczyk, Z. T., and Iyer, R. K. PARM: Adaptive resource allocation for datacenter power capping. In *Workshop on Machine Learning for Systems at NeurIPS 2023*, 2023c.

Qu, G., Wu, H., Li, R., and Jiao, P. DMRO: A deep meta reinforcement learning-based task offloading framework for edge-cloud computing. *IEEE Transactions on Network and Service Management*, 18(3):3448–3459, 2021. doi: 10.1109/TNSM.2021.3087258.

Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S., and Hundt, R. Google-Wide Profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4):65–79, 2010. doi: 10.1109/MM.2010.68.

Rusu, A. A., Rao, D., Sygnowski, J., Vinyals, O., Pascanu, R., Osindero, S., and Hadsell, R. Meta-learning with latent embedding optimization. In *Proceedings of the 7th International Conference on Learning Representations (ICLR 2019)*, 2019. https://openreview.net/pdf?id=BJgklhAcK7.

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T. Meta-learning with memory-augmented neural networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, pp. 1842–1850. JMLR.org, 2016.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

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

Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Provost, J., Simmons, J., Tanda, E., Wanderer, J., Hölzle, U., Stuart, S., and Vahdat, A. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM 2015)*, pp. 183–197, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335423.

Sriraman, A. and Dhanotia, A. 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)*, pp. 733–750, New York, NY, USA, 2020. Association for Computing Machinery.

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

Sun, C., Azari, N., and Turakhia, C. Gallery: A machine learning model management system at Uber. In *International Conference on Extending Database Technology*, 2020.

Sutskever, I., Martens, J., and Hinton, G. E. 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.

Tessler, C., Shpigelman, Y., Dalal, G., Mandelbaum, A., Haritan Kazakov, D., Fuhrer, B., Chechik, G., and Mannor, S. Reinforcement learning for datacenter congestion control. *Proceedings of the 36th AAAI Conference on Artificial Intelligence*, 36(11):12615–12621, Jun. 2022.

Thrun, S. and Pratt, L. *Learning to Learn*. Springer Science & Business Media, 1998.

Tian, H., Liao, X., Zeng, C., Zhang, J., and Chen, K. Spine: An efficient DRL-based congestion control with ultra-low overhead. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies (CoNext 2022)*, pp. 261–275, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450395083.

Van Houdt, G., Mosquera, C., and Nápoles, G. A review on the long short-term memory model. *Artificial Intelligence Review*, 53:5929–5955, 2020.

Vartak, M. and Madden, S. ModelDB: opportunities and challenges in managing machine learning models. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 41(4):16–25, 2018. URL http://sites.computer.org/debull/A18dec/p16.pdf.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS 2017)*, pp. 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Venkataraman, S., Yang, Z., Franklin, M., Recht, B., and Stoica, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (NSDI 2016)*, pp. 363–378, USA, 2016. USENIX Association. ISBN 9781931971294.

Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys 2015)*, Bordeaux, France, 2015.

Verma, V. K., Brahma, D., and Rai, P. Meta-learning for generalized zero-shot learning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, volume 34, pp. 6062–6069, 2020.

Wang, Y., Crankshaw, D., Yadwadkar, N. J., Berger, D., Kozyrakis, C., and Bianchini, R. 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)*, pp. 622–634, New York, NY, USA, 2022a. Association for Computing Machinery.

Wang, Z., Zhu, S., Li, J., Jiang, W., Ramakrishnan, K. K., Zheng, Y., Yan, M., Zhang, X., and Liu, A. X. DeepScaling: Microservices autoscaling for stable cpu utilization in large scale cloud systems. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC 2022)*, pp. 16–30, New York, NY, USA, 2022b. Association for Computing Machinery.

Xia, Z., Zhou, Y., Yan, F. Y., and Jiang, J. Genet: Automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pp. 397–413, New York, NY, USA, 2022. Association for Computing Machinery.

Xue, S., Qu, C., Shi, X., Liao, C., Zhu, S., Tan, X., Ma, L., Wang, S., Wang, S., Hu, Y., Lei, L., Zheng, Y., Li, J., and Zhang, J. 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 (KDD 2022)*, pp. 4290–4299, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393850.

Yadwadkar, N. J., Hariharan, B., Gonzalez, J. E., Smith, B., and Katz, R. H. Selecting the best VM across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC 2017)*, pp. 452–465, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350280.

Yang, A., Lu, C., Li, J., Huang, X., Ji, T., Li, X., and Sheng, Y. Application of meta-learning in cyberspace security: A survey. *Digital Communications and Networks*, 9(1): 67–78, 2023. ISSN 2352-8648. doi: https://doi.org/10.1016/j.dcan.2022.03.007.

Yeganeh-Khaksar, A., Ansari, M., Safari, S., Yari-Karin, S., and Ejlali, A. Ring-DVFS: Reliability-aware reinforcement learning-based DVFS for real-time embedded systems. *IEEE Embedded Systems Letters*, 13(3):146–149, 2021. doi: 10.1109/LES.2020.3033187.

Zhang, K., Wang, P., Gu, N., and Nguyen, T. D. GreenDRL: Managing green datacenters using deep reinforcement learning. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC 2022)*, pp. 445–460, New York, NY, USA, 2022. Association for Computing Machinery.

Zhang, X., Wu, H., Chang, Z., Jin, S., Tan, J., Li, F., Zhang, T., and Cui, B. ResTune: Resource oriented tuning boosted by meta-learning for cloud databases. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD 2021)*, pp. 2102–2114, New York, NY, USA, 2021a. Association for Computing Machinery. ISBN 9781450383431.

Zhang, Y., Goiri, I. n., Chaudhry, G. I., Fonseca, R., Elnikety, S., Delimitrou, C., and Bianchini, R. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021)*, pp. 724–739, New York, NY, USA, 2021b. Association for Computing Machinery.

Zhang, Y., Hua, W., Zhou, Z., Suh, G. E., and Delimitrou, C. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*, pp. 167–181, New York, NY, USA, 2021c. Association for Computing Machinery. ISBN 9781450383172.

Zhang, Z., Liu, Q., Qiu, S., Zhou, S., and Zhang, C. Unknown attack detection based on zero-shot learning. *IEEE Access*, 8:193981–193991, 2020.

Zheng, J., Xu, H., Chen, G., and Dai, H. Minimizing transient congestion during network update in data centers.

In *Proceedings of the 2014 Conference on Emerging Networking EXperiments and Technologies*, pp. 4–6, 2014.

A APPENDICES

A.1 Meta Learner Architecture in FLASH

In this section, we introduce the neural network architecture of FLASH's meta learner (as shown in Figure 11) and the embedding generation process. In FLASH, embeddings are used to explicitly represent and differentiate cloud application and environment pairs (i.e., (A_i, E_j)), and meta-learning enables learning to generate such embeddings in a way that the individuality of each pair can be modeled and extracted. As mentioned in §3.2, the meta learner consists of the input layer, RNN layer, and embedding layer.

Input Layer. The input layer unifies different formats of data samples into a *sequence vector* to be taken by the RNN layer. Let us denote the embedding generation function in meta learner by $f : D \rightarrow \mathbb{R}^d$ where d is the size of the embedding. For supervised learning (SL), the data samples D include a vector of feature variables $[x]$ and labels (i.e., predictions) $[y]$. Each element X_t in the input sequence vector contains the concatenation of x_t and y_t. For reinforcement learning (RL), the data samples D include a set of M RL trajectories generated by the agent interacting with the environment. Each trajectory contains characteristics of the (A_i, E_j) pair in the system management task that the agent is currently managing. Each element X_t in the input sequence vector concatenates the state, action, and reward in each trajectory at time step $t \in [0, T]$ where T is the trajectory length. For instance, as shown in Figure 11, X_0 contains $[S_0(m), A_0(m), R_0(m)]$ from trajectories $m \in [1, M]$, where S_t, A_t, and R_t refer to the state, action, and reward at time step t in each trajectory.

RNN Layer. As mentioned in §3.2, FLASH uses a bidirectional GRU (Gated Recurrent Unit), a special class of RNN (Schuster & Paliwal, 1997), that maintains a high-dimensional hidden state with nonlinear dynamics to acquire, process, and memorize knowledge about each (A_i, E_j) pair. An RNN is a type of neural network that is specialized for processing a sequence of data $X_0, X_1, ..., X_T$ where each indexed element X_t corresponds to one pre-processed variable in the input layer. In particular, we applied a multi-layer, bidirectional gated recurrent unit (GRU) RNN (PyTorch, 2023) to the input sequences. Two unidirectional RNN hidden layers are chained together in opposite directions and act on the same input (as shown in Figure 11). For the forward RNN hidden layer, the first input is X_0, and the last input is X_T, but for the backward RNN hidden layer, the first input is X_T, and the last input is X_0. The output of the bidirectional RNN layer is generated by concatenating together the corresponding outputs (i.e., the hidden states) of the two underlying unidirectional RNN hidden layers. Mathematically, given M input sequences (i.e., RL trajectories), we have the output $O_T = \frac{1}{M}\sum_{m=1}^{M} H_i^m$, where H_i^m is the intermediate output for the m-th trajectory in the i-th (A_i, E_j) pair.

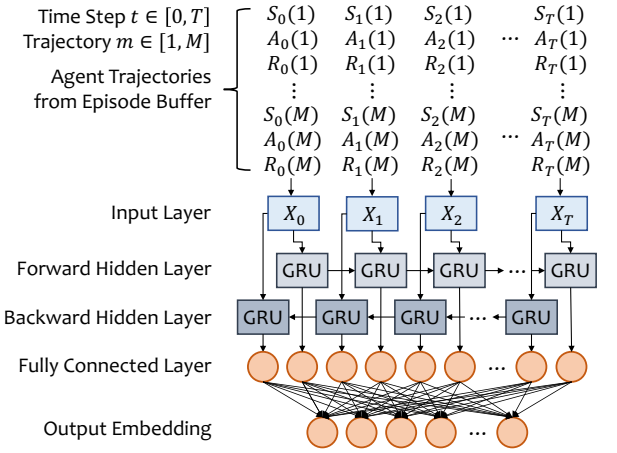


Figure 11. Neural network architecture of FLASH's RNN-based meta learner for embedding generation. *Each variable X_t in the input layer corresponds to a vector of indexed training data samples. In reinforcement learning, $X_t = [s_t(i), a_t(i), r_t(i)], i \in \{1..M\}$ where M is the number of trajectories in the selected episodes buffer. In supervised learning (not shown in the figure), $X_t = [x_t, y_t]$ where x_t and y_t are features and predictions for the t-th sample.*

As mentioned in §3.2, we do not explicitly use the popular memory augmentation technique (Santoro et al., 2016) for the meta learner as the features of our application workloads are not as high-dimensional as those of computer vision tasks (Santoro et al., 2016). We also leave the usage of more advanced sequence models such as long short-term memory (LSTM) (Van Houdt et al., 2020; Hochreiter & Schmidhuber, 1997) and attention-based techniques (e.g., Transformers (Vaswani et al., 2017)) to our future research.

Embedding (FCNN) layer. The output (i.e., a fixed-size vector) from the bidirectional RNN layer is fed to a fully connected neural network (FCNN) layer to generate an embedding that is used to fingerprint/represent the $((A_i, E_j))$ pair with which the base learner is dealing. The input size is equal to the size of the hidden RNN layer, and the output size is equal to d, which is the embedding size. ReLU is used as the activation function. The generated embedding from the FCNN layer will be concatenated by the base learner as part of the feature vector (in the SL case) or the state vector at each time step (in the case of RL).

We implement FLASH's meta learner with PyTorch, and the hyperparameters are shown in Table 4.

A.2 Details of Case Study on Sizeless

Model. Sizeless (Eismann et al., 2021a) uses a fully connected neural network as the predictor for the regression

Table 4. FLASH training hyperparameters.

Parameter	Value
Trajectory Buffer Size	32
Trajectory Expiration Time	300 time steps
Learning Rate	3×10^{-4}
RNN Input Size	256
RNN Hidden Layers	2
RNN Hidden Layer Size	256
Dropout	0.05
Embedding Size	32

task. The features (after feature engineering) and label used in the Sizeless model are shown in Table 5, which are consistent with the original paper (Eismann et al., 2021a). After a grid search to tune the hyperparameters of the model (as shown in Table 6), the final model uses the Adam optimizer, a MAPE loss function, 200 epochs, an L2 regularization of 10^{-2}, and four 256-neuron layers in the neural network.

Applications and Sizeless Dataset. We adopt the 16 representative production cloud workloads selected in Sizeless (Eismann et al., 2021a) based on a survey of 89 industry use cases of serverless computing applications (Eismann et al., 2021b). 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). The Sizeless dataset (released by the original paper) is collected by running 2000 synthetic AWS Lambda applications generated by random sampling with replacement from the segment pool (consisting of the 16 selected representative production application segments) and combining the selected segments together. Each segment represents the smallest granularity of common workloads in cloud datacenters.

The original Sizeless dataset includes measurements on the execution time and resource consumption metrics (see (Eismann et al., 2021a) for a full table of dataset columns) for all applications across six different memory sizes (128 MB, 256 MB, 512 MB, 1024 MB, 2048 MB, 3008 MB) for ten minutes each at 30 requests per second with an exponentially distributed inter-arrival time. In the future, the number of implemented segments can easily be extended if specific workload profiles are missing.

To study the model prediction accuracy degradation when encountering new applications or compute platform changes, we constructed two new datasets, **OpenWhisk** and **CloudBandit**. We first implemented a synthetic application generator based on the open-sourced AWS Lambda application generator from Sizeless (Eismann et al., 2021a). Overall, we generated 1000 unique applications that are deployable on both OpenWhisk (for evaluation of the resource configuration search task) and Kubernetes (for evaluation of the workload autoscaling task in Appendix A.3 and CPU frequency scaling task in Appendix A.4).

Table 5. Features and labels in Sizeless.

Features (X)
Base memory, Execution time under the base memory, Heap used, User CPU time, System CPU time, Voluntary context switches, Bytes written to file system, and Bytes received over network, Target memory
Label (y)
Execution time under the target memory

Table 6. Sizeless training hyperparameters.

Parameter	Parameter Range	Selected
Optimizer	SGD, Adam, Adagrad	Adam
Loss	MSE, MAE, MAPE	MAPE
Epochs	200, 500, 1000	200
Neurons	64, 128, 256	256
L2	0, 0.0001, 0.001, 0.01	0.01
Layers	2, 3, 4, 5	4

OpenWhisk Dataset. Following the same dataset collection methodology as Sizeless (Eismann et al., 2021a), we deployed each generated synthetic application on a 50-VM OpenWhisk cluster setup on IBM Cloud. In addition to the memory size of the function container (which is the only resource configuration considered in Sizeless), we also consider CPU allocation (i.e., `cpu.shares` used in OpenWhisk) as another resource configuration. The collected metrics remain the same as in the Sizeless dataset.

CloudBandit Dataset. The CloudBandit dataset (Lazuka et al., 2022) covers application resource configuration (i.e., number of nodes, CPU family type, number of vCPUs, and VM type), performance metrics, and system metrics on three different public cloud platforms. The CloudBandit dataset was originally collected by running 30 production workloads on a variety of different resource configurations across three different cloud providers: Amazon Web Services, Microsoft Azure, and Google Cloud Platform. We preprocessed the dataset to align it with the Sizeless model training dataset by replacing the target memory size (used in the Sizeless dataset) with the target resource configurations, such as the VM type and vCPU count (used in the CloudBandit dataset).

Datasets OpenWhisk and CloudBandit are used to evaluate the ML model's generalizability across different applications, while the dataset CloudBandit is also used to evaluate the model's generalizability across different computing infrastructures. Results are presented in §2.2 and §5.1.

A.3 Details of Case Study on FIRM

Model. FIRM (Qiu et al., 2020) uses an actor-critic RL algorithm, DDPG (Lillicrap et al., 2016). The RL agent monitors the system- and application-specific measurements and learns how to scale the allocated resources vertically and horizontally. Table 7 shows the agent's state and action spaces. Table 8 shows the model hyperparameters. The goal is to achieve high resource utilization (RU) while maintain-

Table 7. RL state-action space and reward function in FIRM (Qiu et al., 2020) for the workload autoscaling task.

State Space (s_t)
Resource Limits (CPU, RAM), Resource Utilization (CPU, Memory), SLO Preservation Ratio (Latency, Throughput), Observed Load Changes
Action Space (a_t)
Resource Limits (CPU, RAM), Number of Replicas
Reward Function (r_t)
$r_t = \alpha \cdot SP_t + (1-\alpha) \cdot (RU_{cpu} + RU_{memory})/2$

Table 8. FIRM 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})

ing 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, $SP = 1$. The reward function is then defined as $r_t = \alpha \cdot SP_t \cdot |\mathcal{R}| + (1-\alpha) \cdot \sum_{i \in \mathcal{R}} RU_i$, where $\mathcal{R}$ is the set of resources (i.e., container CPU limit and memory capacity in our case). 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).

Applications and Traces. As mentioned in Appendix A.2, we generated 1000 synthetic applications (deployable on both OpenWhisk and Kubernetes) using the selected 16 representative production cloud serverless workloads as application segments (same as in the resource configuration search task mentioned in Appendix A.2). We reuse these application segments in the task of workload autoscaling as well because serverless workloads are highly dynamic (and thus require autoscaling) and rely on the provider to manage the resources. For RL agent training and inference, we use real-world datacenter traces (Zhang et al., 2021b) released by Microsoft Azure, collected over two weeks in 2021. Next, we deploy the selected workloads as Deployments in a five-node Kubernetes cluster on IBM Cloud Virtual Private Cloud (VPC) and run an RL-based multi-dimensional autoscaler with each Deployment, controlling both the number of replicas (horizontal scaling) and the container sizes (vertical scaling). All nodes run Ubuntu 18.04 with four cores, 16 GB memory, and a 200 GB disk.

Application Updates/Patches. We introduce, in total, seven scenarios to explore model performance degradation when facing application patches, service payload size changes, or load pattern variations: (1) 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 is changed from [128 KB, 256 KB, 384 KB] to [512 KB, 768 KB, 1024 KB]. (2) For database services, the size of the database table being scanned is changed from 1024 items to 10240 items. (3) For floating-point number calculation, the number of operations is changed from 10^8 to 20^8. (4) For image manipulations, the image dimension is changed from 40×40 to 160×160. (5) For text processing, the JSON file size is changed from [250 B, 500 B, 1 KB] to [2 KB, 3 KB, 5 KB]. (6) For ML model serving, we change the matrix multiplication dimension from 50 to 150. (7) For load pattern changes, we divide 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 load ($\leq 10^5$ per day).

Table 9. RL state-action space and reward function in SmartOverclock (Wang et al., 2022a) for CPU frequency scaling.

State Space (s_t)
Instructions per second (IPS), CPU usage, Measured core frequency, SLO Preservation Ratio (Latency, Throughput)
Action Space (a_t)
CPU core frequency (every second)
Reward Function (r_t)
$r_t = \alpha \cdot ((IPS_t - IPS_{t-1})/IPS_t)_{\Delta freq>0} + (1-\alpha) \cdot SP_t$

Table 10. SmartOverclock (Q-Learning) training hyperparameters (adopted from Stable Baseline3 (Baseline3, 2023)).

Parameter	**Value**
Learning Rate	1×10^{-4}
Learning Starts	50000
Buffer Size	1000000
Batch Size	32
Discount Factor	0.99
Target Network Update Rate	1.0
Exploration Fraction	0.1

A.4 Details of Case Study on SmartOverclock

Model. SmartOverclock (Wang et al., 2022a) is an on-node learning-based agent developed by Microsoft to adjust the CPU core frequency of a running VM dynamically. SmartOverclock uses an RL algorithm called Q-Learning (Baseline3, 2023). At each time step t (every 1-second interval), the agent monitors the average Instructions Per Second (IPS) performance counter across the cores of each VM and learns when to adjust the core frequency. Table 9 shows the model's state and action space. Table 10 shows the model training hyperparameters.

The state vector includes IPS, CPU usage, and current core frequency that are measured at each time step t. For VMs with SLOs or measurable application-level metrics, the same variable SLO preservation ratio (SP_t) is also considered. Based on the state, the agent picks the fre-

quency for the next time epoch. To balance the performance improvements with the extra power cost (of increasing the core frequency), the reward function is defined as $r_t = \alpha \cdot ((IPS_t - IPS_{t-1})/IPS_t)_{\Delta freq>0} + (1-\alpha) \cdot SP_t$, with the assumption that workload benefits from a higher frequency when (a) higher CPU frequencies increase the IPS, or (b) the SLO preservation ratio is high.

Applications and Environments. We reuse the application workloads and application patches described in Appendix A.3 to evaluate model adaptability for new applications or application updates. To evaluate the model adaptability on heterogeneous compute infrastructures, we run experiments on three types of processors: (1) an Intel Xeon E5-2683 v3 2.0 GHz processor, which is capable of running up to 3.0 GHz, (2) an Intel Xeon CPU E5-2695 v4 2.1 GHz processor, which is capable of running up to 3.1 GHz, and (3) an AMD EPYC 7302P 3.0 GHz processor, which is capable of running up to 3.3 GHz.

A.5 Additional Motivating Examples

Congestion control agents at the transport layer adjust the sending rate based on the measured network statistics. Prior work (Tessler et al., 2022; Jay et al., 2019; Ma et al., 2022; Tian et al., 2022) has proposed RL-based solutions. For instance, Aurora (Jay et al., 2019) decides the sending rate at the beginning of each time step (of length proportional to RTT) to maximize the reward (a combination of throughput, latency, and packet loss rate). RL agent trains a policy to optimize the reward over a given distribution of training network environments (e.g., network connections with certain bandwidth patterns, delay, and queue length). It is hypothesized that local history contains information about patterns in traffic and network conditions and thus can be exploited for better rate selection by learning the mapping from experience via a deep RL approach. For example, the learned RL agent is able to distinguish non-congestion loss from congestion-induced loss, while TCP CUBIC halves the sending rate upon any occurrence of loss, and thus fails to fully utilize the link's bandwidth (Jay et al., 2019).

However, training in a wide range of network environments leads to suboptimal performance, whereas training in a narrow distribution of environments results in poor generalization. For the distributions spanning a wide variety of network environments (e.g., a large range of possible bandwidth or link delay), an RL policy may perform poorly when tested in a different environment than the set of environments seen during training. We generate simulated training environments with various network condition parameters following prior work (Mao et al., 2019a; Jay et al., 2019; Xia et al., 2022) (in total 625 environments). The parameters are used as configurations in the network simulator to specify the network condition, such as bandwidth range, link delay, queue size, random loss rates, and delay noises.

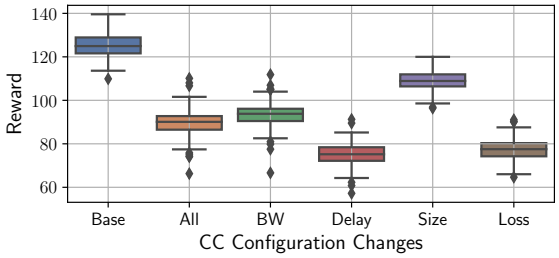


Figure 12. Reward degradation in Aurora. *X-axis represents different testing scenarios of network environment changes. In "Base", the RL agent is trained and tested in the same environment. In "BW", "Delay", "Size", and "Loss", the RL agent is tested in different environments regarding bandwidth, link delay, queue size, and loss rate, respectively. In "All", the RL agent is tested in all different scenarios.*

We take the open-source implementation of the RL model in Aurora to explore the RL agent performance degradation when encountering network environments different from the one used for training. Figure 12 shows the per-episode reward of the Aurora agent in different scenarios of network environment changes. We find that the performance degradation of the RL agents (regarding the reward) can be up to 32.8% at the 90th percentile and have a median degradation of 27%. In addition, we find that among all configurations, changes in queue size have the least effect on the agent performance (with an average of 12.5% reward degradation), while link delay changes have the highest (40.2%).

CPU frequency scaling also benefits from RL-based solutions (Wang et al., 2022a; Zhang et al., 2022; Yeganeh-Khaksar et al., 2021; Chen & Marculescu, 2015; Islam & Lin, 2017) where the agent adjusts the frequency of CPU cores to balance application performance improvements with the extra power cost. For example, SmartOverclock (Wang et al., 2022a) leverages an RL model Q-Learning to pick the frequency at each time step based on the average Instructions Per Second (IPS) performance counter and the current frequency across the cores of each VM. However, heterogeneous workloads have different scaling factors (i.e., the sensitivity of performance benefit given frequency increase), and various processors offer quite different scaling ranges and turbo performance. We explore the effect of changes in both workloads and processor types (as described in Appendix A.4) on the RL agent performance. Figure 13 shows that the per-episode reward degradation of the RL agents can be up to 44.2% at the 90th percentile when testing on a different set of workload changes and 17.3% when testing on a different processor.

A.6 Additional Case Study on Congestion Control

We also integrate FLASH with a congestion control agent at the transport layer selecting the sending rate based on the sender's observations of the real-time network conditions.

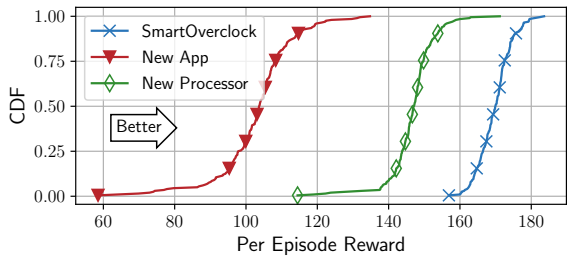


Figure 13. Reward degradation in SmartOverclock.

As mentioned in Appendix A.5, in RL-based congestion control, the agent is trained to learn a policy to optimize performance over a given distribution of training network environments. If the distribution changes for new network environments, the RL agent can fail to adapt quickly because it is not trained to generalize.

RL Formulation and Implementation. We take the open-source implementation of Aurora, an RL-based congestion control agent from prior work (Jay et al., 2019), as the base learner to integrate with FLASH. In Aurora's RL formulation, the agent maps a locally perceived history of feedback from the traffic receiver, which reflects past traffic and network conditions, to the next choice of sending rate at the traffic sender. Aurora (Jay et al., 2019) uses an actor-critic RL algorithm, PPO (Schulman et al., 2017). Table 11 shows the state and action spaces of the RL model. Table 12 shows the model training hyperparameters.

In the RL formulation of the congestion control task, the RL agent is on the traffic sender side, and its actions translate to changes in sending rates. At each time step t, the sender can adjust its sending rate x_t, which then remains fixed throughout the time window until time step $t+1$. RL states are bounded histories of network statistics that are observed by sending packets at a rate x_t and receiving packet acknowledgments. In summary, the network statistics vectors consist of (a) latency gradient/inflation, the derivative of latency with respect to time; (b) latency ratio, the ratio of the current mean latency to the minimum observed mean latency in the connection's history; and (c) sending ratio, the ratio of packets sent to packets acknowledged by the receiver. The reward function is defined as $r_t = \alpha \cdot Throughput_t + \beta \cdot Latency_t + \gamma \cdot Loss_t$ where throughput is measured in packets per second, latency in seconds, and loss is the proportion of all packets sent but not yet acknowledged at time step t.

We adopt the same model design and hyperparameters as used in the open-source implementation of Aurora. The resultant agent is referred to as **FLASH-Aurora**. Similar to the base learner implemented for FIRM (as described in §4), the base learner sends RL trajectories by calling `InsertDataSamples([<S, A, R>])` after each RL episode. The embedding is then retrieved by calling `GetEmbedding()` and appended to the state vector (which is then taken by the actor network of PPO to generate the final actions) in the base learner.

Table 11. RL state-action space and reward function in Autora (Jay et al., 2019) for the congestion control task.

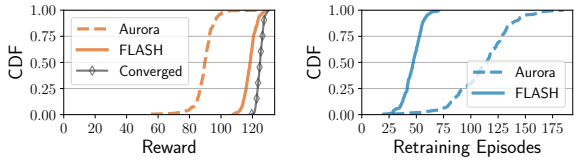
State Space (s_t)
Sending/receiving rate, Sending/receiving duration, avg RTT in a time window, min RTT, RTT inflation, RTT ratio, Ack/Sent latency inflation, Loss/Sent ratio
Action Space (a_t)
Sending rate in the current time window
Reward Function (r_t)
$r_t = \alpha \cdot Throughput_t + \beta \cdot Latency_t + \gamma \cdot Loss_t$

Table 12. Aurora training hyperparameters.

Parameter	**Value**
# Time Steps per Episode	2048 × 64 mini-batches
Learning Rate	3×10^{-4}
Discount Factor	0.99
GAE Lambda	0.95
CLIP Range	0.2
Entropy Coefficient	0.005
Value Function Coefficient	0.5

Evaluation Setup. We train and test FLASH-Aurora in a network environment simulator (Jay et al., 2019) that can simulate network links with a wide variety of network scenarios. The simulator comes with a range of configurations: link bandwidth (in Mbps), link latency or RTT (in ms), packet queue size, and package loss rate. In the original paper, the configuration of the training environment is selected uniformly at random from ranges of parameters that are set to [1.2 Mbps, 6 Mbps] for link bandwidth, [50 ms, 500 ms] for link latency, [0%, 5%] for loss rate, and [2, 2981] for queue size. To train and evaluate RL agents in a wider variety of network scenarios, we expand the ranges of configuration parameters to [0.1 Mbps, 128 Mbps] for link bandwidth, [1 ms, 512 ms] for link latency, [0%, 10%] for loss rate, and [1, 10240] for queue size. We then divide the range for each configuration into five sub-ranges, and each simulation environment is constructed by randomly selecting the configuration parameter range from the five sub-ranges, resulting in 625 environments in total. In the evaluation, we repeat five experiment runs wherein each run, the pre-training of FLASH-Aurora is carried out in 200 randomly selected environments (i.e., the "pre-training pool" in Figure 6), and the adaptation evaluation is done on the remaining 425 environments (i.e., "adaptation pool").

Reward Drop without Adaptation. To evaluate the performance degradation without retraining, we design a congestion control A/B test where FLASH-Aurora agent and the original Aurora agent are the two variants controlling the congestion control for the same set of traces in the simulator. We repeat the A/B test 100 times. In each test, we randomly select an environment from the adaptation pool and train



(a) Reward Degradation. *(b)* Retraining Cost.

Figure 14. Comparison of the performance and retraining cost of the Aurora model with FLASH.

the agent until convergence. We then randomly select ten other different environments from the adaptation pool for reward drop evaluation (i.e., RL policy-serving). Figure 14(a) shows the CDF of the per-episode reward, and we find that FLASH-Aurora reduces the average reward drop percentage from the baseline (i.e., the agent trained to convergence on the testing environment), labeled as "Converged" in Figure 14(a), from 28% to 5.6%.

Model Adaptation Cost. We compare FLASH with the transfer learning (TL)-based approach to retrain an RL agent for a new network environment based on previous RL experience gained for known environments. In the TL-based approach, the model parameters (weights) are shared between the agents managing the known environments and the new environment. We measure the retraining time (to convergence) of Aurora (TL) and FLASH-Aurora. The results are shown in Figure 14(b). We find that, on average, FLASH-Aurora adapts 2.4× faster than TL, with a 58.3% reduction in RL episodes required to convergence (on average 112 episodes compared to 46.7 episodes).

A.7 Amortized Training Cost Analysis

As discussed in §7, pre-training across a distribution of (A_i, E_j) pairs can require a large amount of training overhead (e.g., pre-training on 200 pairs costs 5.2 hours with an NVIDIA Tesla V100 (16 GB) GPU). However, the retraining/adaptation of a base learner for each novel (A_i, E_j) pair from a potentially larger-scale adaption pool only requires no or fewer model update iterations. In addition, such lightweight fine-tuning at scale can amortize the initial pretraining cost and thus reduce the per (A_i, E_j) pair training cost. This trade-off allows us to reduce the overall training effort, especially in scenarios where the application or environment is dynamic or constantly evolving.

To understand the amortization of the training cost for different scales of adaptation pool of (A_i, E_j) pairs, we take the workload autoscaling task as an example, pre-train FLASH on different sizes of the pretraining pool of applications (i.e., 50, 100, 150, and 200 applications), and evaluate the adaptation cost (i.e., number of RL training episodes) when adapting to different number of novel applications. We compare FLASH with transfer learning (TL). In FLASH, the randomly initialized model is pre-trained on the pre-training pool and then used for adaptation to each novel

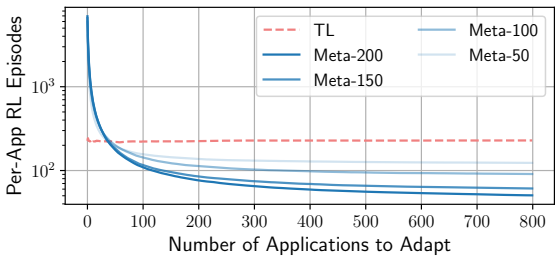


Figure 15. Training cost analysis of transfer learning (TL) and meta-learning with various sizes of pretraining application pool in the task of workload autoscaling. *Y-axis is in log scale and shows the per-application adaptation cost regarding RL episodes and X-axis shows the number of applications that the model has been adapted to.*

application. In TL, the randomly initialized model (i.e., FIRM) is continuously being trained on each novel application (without any pre-training). We show in Figure 15 the training cost analysis of FLASH and TL with different sizes of the pretraining application pool in the task of workload autoscaling. X-axis represents the number of applications that have been adapted to, and Y-axis represents the average per-application RL training episodes across all applications that have been adapted to.

We find that with the size of the initial pretraining pool growing, the initial pretraining cost increases (but the increase rate slows down), while the adaptation cost in fine-tuning decreases. For instance, pre-training FLASH on 200 and 100 applications costs 6900 and 5800 episodes, respectively, while the average adaption cost for fine-tuning is 41.8 and 83.5 episodes, respectively. In terms of per-application adaptation cost, compared to transfer learning (labeled as "TL" in Figure 15), FLASH has a higher cost when only using it to adapt to a smaller number of unseen applications. However, when FLASH has been used to adapt to more than 60 applications, the per-application adaptation cost starts to be lower than TL, due to the amortization of the pre-training cost with the benefits of lightweight fine-tuning.