On the Promise and Challenges of Foundation Models for Learning-based Cloud Systems Management

Haoran Qiu[1] Weichao Mao[1] Chen Wang[2] Hubertus Franke[2] Zbigniew T. Kalbarczyk[1]
Tamer Başar[1] Ravishankar K. Iyer[1]
[1]*University of Illinois Urbana-Champaign* [2]*IBM Research*

Abstract

Foundation models (FMs) are machine learning models that are trained broadly on large-scale data and can be adapted to a set of downstream tasks via fine-tuning, few-shot learning, or even zero-shot learning. Despite the successes of FMs in the language and vision domain, we have yet to see an attempt to develop FMs for cloud systems management (or known as cloud intelligence/AIOps). In this work, we explore the opportunities of developing FMs for cloud systems management. We propose an initial FM design (i.e., the FLASH framework) based on meta-learning and demonstrate its usage in the task of resource configuration search and workload autoscaling. Preliminary results show that FLASH achieves 52.3–90.5% less performance degradation with no adaptation and provides 5.5× faster adaptation. We conclude this paper by discussing the unique risks and challenges of developing FMs for cloud systems management.

1 Introduction

Lately, foundation models (FMs) have exhibited notable efficacy in tackling a range of intricate tasks, especially in natural language processing (NLP) or computer vision. An FM is typically referred to as any ML model that is trained on broad data at scale and can be adapted to a wide range of downstream tasks [3]. The training of such an FM is called *pretraining* and the adaptation to a downstream task is called *fine-tuning*. For example, a pre-trained BERT model [10] (an FM in the NLP domain) has shown 6–11% improvement in downstream tasks including question answering, sentiment analysis, and semantic pairing with light fine-tuning (i.e., 2–4 epochs on much smaller fine-tuning datasets).

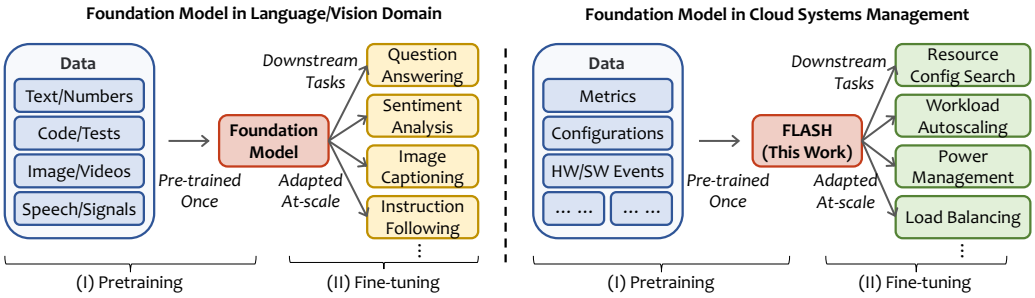


Figure 1: Foundation models in language/vision tasks (left) and the idea of FLASH (right).

FMs have been driving a paradigm shift in the way that modern-day ML models are trained and used. As illustrated in Fig. 1, rather than learning each task-specific model from scratch (which is costly in terms of both dataset collection and training time), an FM model that is pre-trained once can be adapted to various tasks via lightweight fine-tuning [7] or few/zero-shot learning [57, 66].

Despite the successes of FMs in the language and vision domains, there is relatively little work exploring the development of an analogous FM for *cloud systems management*, also known as cloud

Machine Learning for Systems Workshop at 37th NeurIPS Conference, 2023, New Orleans.

intelligence or AIOps [2, 9]. Over the years, numerous ML-based solutions (especially supervised learning and reinforcement learning or RL) have been developed and deployed by cloud providers to build, operate, and optimize cloud systems [2, 8, 14, 26, 21]. The tasks that benefit from ML include resource management [38, 20, 62, 40, 59, 43, 56, 33, 61, 22, 69, 11, 42], job scheduling [30, 31, 64, 1, 5, 32], congestion control [60, 49, 19, 25, 28, 50], and power management [58, 65, 63, 6, 18].

However, the existing way of developing and training one model per task is costly in terms of both training time and dataset collection effort. Yet, the learned model in each task requires substantial retraining (even with transfer learning) when serving new applications or cloud infrastructures.

In this position paper, we lay the groundwork for developing FMs in cloud systems management. We first showcase a need for FMs in two systems management tasks, namely, resource configuration search and workload autoscaling. We then propose FLASH, an initial design of an FM with meta-learning, and demonstrate the usage of FLASH in the two aforementioned tasks. Finally, we discuss some potential risks and challenges that must be considered when developing such general-purpose models for cloud systems management, which we believe will foster future research in this domain.

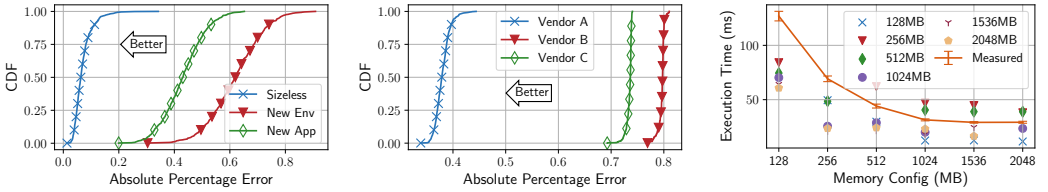
2 FLASH: An Initial Approach to FM

2.1 Case Studies and Motivation

In this section, we describe two ML-for-systems tasks, namely, *resource configuration search* (with supervised learning) and *workload autoscaling* (with RL), and illustrate the motivation for an FM.

• ***Resource configuration search*** [56, 33, 61, 22, 69, 11] is a critical cloud systems management task to decide the optimal resource allocation for VMs/containers to run (e.g., the memory size for a serverless function) to meet application requirements without overprovisioning. This task is usually modeled as a regression problem where a supervised learner is trained to predict the performance and/or cost given a resource configuration. For example, Sizeless [11] leverages a fully connected neural network to predict the average execution times of a serverless function given a target memory size based on the monitoring data when running with a base memory size.
• ***Workload autoscaling*** [38, 20, 62, 40, 59, 43, 41] is modeled as a sequential decision-making process, i.e., to decide the vertical and/or horizontal concurrency of the controlled application. RL is well-suited for learning such policies, as it provides a tight feedback loop for exploring the state-action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules) [30, 40]. For example, FIRM [38] leverages an RL model DDPG to scale vertically (i.e., CPU/memory allocation) and horizontally (i.e., number of replicas). The goal of the RL agent is to maximize resource utilization while maintaining application service-level objectives (SLOs).

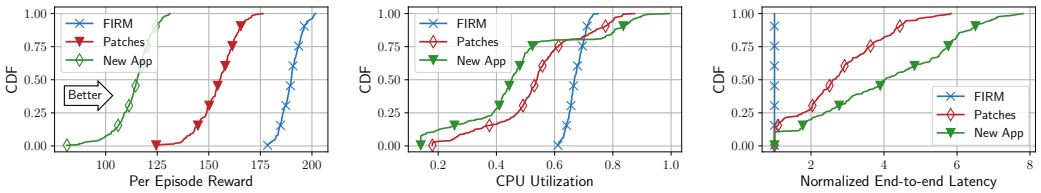
Adaptation Requirements. Unfortunately, application or cloud environment heterogeneity could invalidate model assumptions and result in suboptimality for the trained model. Re-training models can be costly and requires a large amount of training data [26, 1]. Specifically, we took the open-source implementation of Sizeless and FIRM models, trained them in the original setup described in their papers, and studied the model performance degradation on new setups. Detailed descriptions of the model architecture, training, and datasets are deferred to Appendices A and B.



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

Figure 2: Performance of trained Sizeless model [11] on unseen cloud platforms or new applications.

For Sizeless, we quantify the model performance degradation and show in Fig. 2 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 [11]). When testing the trained model on unseen applications deployed on the same OpenWhisk platform, the CDF (labeled as "New App") shows a 9.9× increase in median APE. When testing the trained model (using data from cloud vendor A) on the same applications but running on



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

Figure 3: Performance of trained FIRM model [38] on unseen applications or application updates.

vendor B or C, the CDF (labeled as "New Env") shows a 16× increase in median APE. Fig. 2 (c) shows an example of misprediction when testing on an unseen application (i.e., Airline Booking). As we can see, using different base memory configurations, the prediction can be under-/over-estimated, leading to either application performance degradation or unnecessarily higher deployment costs.

Fig. 3(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 investigated the degradation regarding container CPU utilization and the 99th percentile application end-to-end latency. Fig. 3(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 workloads' heterogeneous impacts on CPU utilizations and varying performance sensitivity to resource allocations [38]. The learned mapping between RL states and optimal actions is no longer valid and thus requires retraining.

Key Takeaway: *The characterization results highlight significant performance degradation when applying either supervised learning or RL approaches to new applications and environments, which emphasizes the need for innovative approaches to address the adaptability and efficiency of cloud systems management models. To avoid high (re-)training costs and enable fast adaptation, a promising direction is to leverage the emerging "pretrain-finetune" paradigm introduced by FMs. Rather than developing and training one model per task for a specific application and cloud environment setup, we aim to pre-train an FM that provides (1) fast model adaptation to new applications and environments, and (2) a common basis upon which many task-specific models are built via adaptation.*

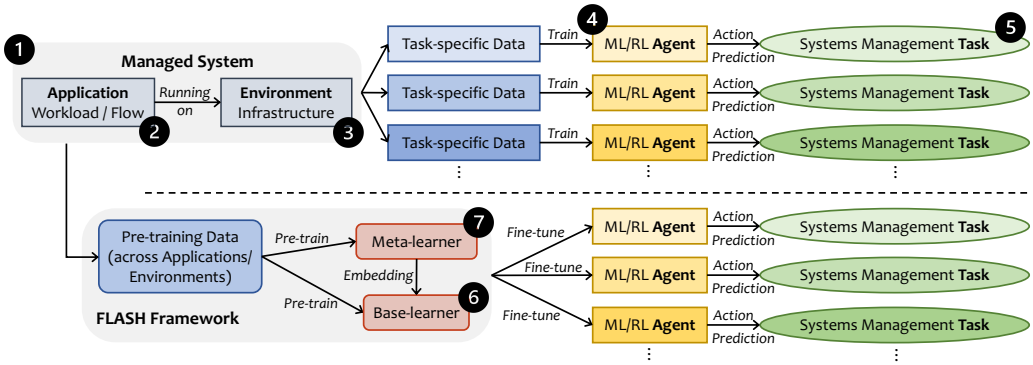


Figure 4: Workflow of FLASH in transforming traditional ML model development processes.

2.2 FLASH Design

To facilitate efficient and lightweight model adaptation, we propose the design of FLASH, which brings the *pretrain-finetune* paradigm of FMs to cloud systems management. Fig. 4 illustrates how FLASH transforms the traditional ML model development for any specific systems management task. Given a systems management task ❺ in a managed system ❶ (e.g., workload autoscaling in a Kubernetes cluster), there are three main components: application ❷ (i.e., the deployed workload), environment ❸ (i.e., the underlying infrastructure), and ML/RL agent ❹ (the model trained to control the task). Instead of developing and training one model per task for each ⟨`app`, `env`⟩ pair, FLASH provides a framework for pre-training a base model as the common basis with *meta-learning* and fine-tuning the base model to adapt to any ⟨`app`, `env`⟩ pair in the task.

We introduce *embedding*-based meta-learning [44, 35] in FLASH where the ML/RL agent is modeled as a *base-learner* (❻ in Fig. 4) and a *meta-learner* (❼)[1] is designed for learning to generalize across applications and environments. The base-learner discovers policies that adapt to a specific application and intra-environment dynamicity for each ⟨app, env⟩ pair, as each ML/RL agent originally does. Meanwhile, the meta-learner generalizes across ⟨app, env⟩ pairs to address inter-environment dynamicity and application heterogeneity. Inspired by *word embeddings* in NLP [34, 52], where individual words are represented as numerical vectors in a lower-dimensional space and words with similar semantic meanings get similar representations, FLASH's meta-learner generates an *embedding* that projects the application/environment-specific characteristics to a vector space. On this projected vector space, FLASH maps ⟨app, env⟩ pairs with similar characteristics (e.g., applications with similar performance sensitivity to resource contention) to neighboring locations (similar to "clustering"), while projecting those with quite different characteristics to different clusters.

Interaction between Meta-learner and Base-learner. We focus on supervised learning (SL) and RL which are the most common ML categories used in the systems domain [29]. The meta-learner samples labeled data points (in SL) or RL trajectories and generates an embedding that accurately represents the application running in the environment. As shown in Fig. 5, we use a bidirectional GRU (a special class of RNNs) [48, 15, 46] for embedding generation while leaving the exploration of more advanced sequence models such as LSTMs [53, 16] and attention-based techniques (e.g., Transformers [55]) to our future research. 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 workloads and computing infrastructure changes. See Appendix C for details about the meta-learner architecture.

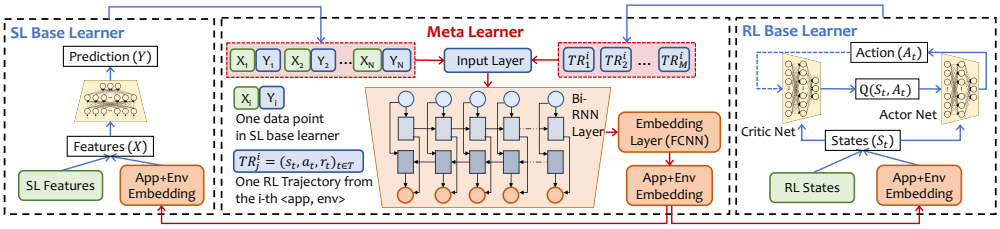


Figure 5: Interaction between the base-learner and the meta-learner in FLASH.

Pre-training and Fine-tuning with FLASH. During pre-training, FLASH is exposed to a pool of ⟨app, env⟩ pairs and trained to discriminate the *individuality* of each pair with meta-learning. For *commonality*, FLASH also learns a *shared model* via the base-learner that can be used as a common basis to build and adapt to customized models across different ⟨app, env⟩ pairs. *The resulting meta-learner and the shared model in* FLASH *are analogous to a pre-trained FM.*

During fine-tuning, since the shared model is conditioned on application-/environment-specific embeddings, the adaptation process only requires limited exposure. The embedding generation corresponds to a few SL samples or RL trajectories to feed to the input layer of the meta-learner. Note that even though the new pair has never been encountered, adaptation is still possible when the new pair shares similar patterns with the encountered ones [35, 13, 36, 17]. For the pairs with quite dissimilar patterns (detected based on the distance in the embedding space), the shared model/policy conditioned on the embedding can be further fine-tuned to adapt to the optimal customized model/policy. *Overall, the pretrain-finetune paradigm provides an efficient way to balance the cost of pre-training with the need for fast adaptation for heterogeneous cloud applications and environments.*

3 Evaluation

We evaluate (a) the adaptation efficacy of FLASH across diverse applications and environments in two case studies (described in §2.1), and (b) the training and inference overhead of FLASH.

Resource Configuration Search. We implemented the Sizeless model with and without FLASH in the task of resource configuration search, i.e., predicting the application runtime given the target resource configuration. Three datasets were used for training and evaluation: (a) the Sizeless dataset

[1]The meta-learner together with a shared model pre-trained via the base-learner compose the core of FLASH, which is referred to as an FM that can fine-tuned to customize for particular applications or environments.

open-sourced by the original paper [11] consisting of 2000 serverless applications, (b) the OpenWhisk dataset, which we collected on a local OpenWhisk cluster following the same methodology as [11], and (c) the CloudBandit dataset [23] for 30 VM-based applications on three public cloud platforms. Datasets (a) and (b) are used to evaluate the adaptability across different applications, while dataset (c) is used to evaluate the adaptability across different cloud computing infrastructures. More details about the task and datasets in this case study are deferred to Appendix A.

Table 1: The prediction error of a Sizeless agent (i.e., MAPE) with and without FLASH. *The number of samples used as the base configuration to predict the target configuration is indicated as X-shot.*

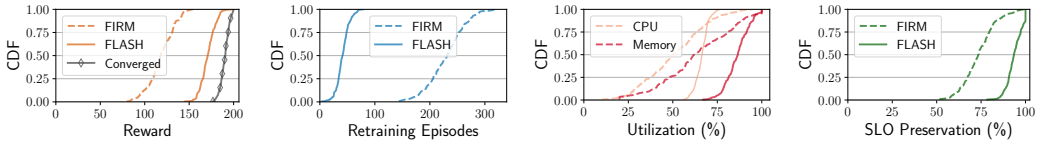
Dataset	Sizeless [11]			OpenWhisk			CloudBandit [23]		
# of Samples	1-shot	2-shot	3-shot	1-shot	2-shot	3-shot	1-shot	2-shot	3-shot
Sizeless (training)	0.040	0.036	0.035	0.316	0.258	0.236	0.610	0.439	0.424
Sizeless (testing)	0.360	0.400	0.336	0.823	0.552	0.540	0.985	0.889	0.798
FLASH-Sizeless (training)	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%

Table 1 shows the model performance comparison where the evaluation metric is mean absolute percentage error (MAPE). For agents trained using the Sizeless dataset and OpenWhisk dataset (both serverless applications), we use four real-world applications from [11] deployed on the OpenWhisk cluster as the testing dataset. For agents trained using the CloudBandit dataset, we randomly split the dataset by 8:2 to separate the testing dataset (20%) from the meta-learning training dataset. We evaluate the model performance on previously unseen applications with 1-shot, 2-shot, and 3-shot configurations, corresponding to the number of data samples the model uses as the base configuration(s) to predict the application performance under the target configuration. In Sizeless, OpenWhisk, and CloudBandit datasets, FLASH helps achieve up to 90.5%, 56.6%, and 52.31% lower MAPE compared to the original Sizeless model, respectively. The improvement on the CloudBandit dataset is the lowest because both Sizeless and OpenWhisk datasets contain detailed system-level container-related metrics (e.g., heap used, user/system CPU time, voluntary context switches, and bytes written to the file system), while the CloudBandit dataset does not. Therefore, the embeddings generated from only performance metrics and resource configurations are not as expressive as the embeddings generated with additional system-level metrics during application runtimes.

Workload Autoscaling. We generated 1000 synthetic applications based on the open-source application generators and serverless benchmarks in Sizeless [11] as 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 used real-world datacenter traces [68] released by Microsoft Azure, collected over two weeks in 2021. Next, we deployed the selected workloads as Deployments in a five-node Kubernetes cluster in a public cloud and ran an RL-based multi-dimensional autoscaler with each Deployment, controlling both the number of replicas (horizontal scaling) and the container sizes (vertical scaling). We divided the 1000 generated applications with an 8:2 ratio. The 800 applications are used in pre-training while the remaining 200 applications are used to evaluate the adaptability in fine-tuning. The total application runtime is $\sim$60 days, and the offline pre-training time is $\sim$5.2 hours with an NVIDIA Tesla V100 (16 GB) GPU. More details are deferred to Appendix B.

To evaluate the adaptation cost, we perform A/B tests (100 times) on FIRM with and without FLASH. In each test, we randomly select a workload from the application pool and train the FIRM agent until convergence. We then randomly select ten other different workloads from the pool for RL reward drop evaluation. Fig. 6 shows the performance degradation without adaptation as well as the adaptation costs. FLASH reduces the average reward drop percentage from the baseline (i.e., FIRM agent trained using the testing applications) from 37% to 10.5%. FIRM leverages transfer learning (TL) with parameter sharing to retrain an RL agent for a new application based on previous RL experience gained when previously training the RL agent for known applications [38]. We measure the retraining time, utilization deficit (compared to the converged RL policy), and performance degradation (compared to SLOs) of the TL-based approach and FLASH. We find that FLASH adapts 5.5× faster than TL, which results in 67.1% less performance degradation and 4.6× less CPU utilization deficit, during retraining.

Training and Inference Overhead. We evaluate the overhead of FLASH introduces in each case study. At the *inference* stage, generating embeddings and including the embedding in the forward pass process of the original neural network introduce additional latency. The average inference overhead is up to 3.5× (i.e., from 2.3 ms to 8.1 ms for FIRM). However, compared to the second



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

Figure 6: Comparison of the performance and retraining cost of the FIRM model with FLASH.

(s)-level RL time steps, such 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 is negligible compared to the model update latency of the original ML/RL agent (1.25 s compared to 1.21s).

4 Discussion and Open Questions

Why Meta-learning for FMs? We observe a growing paradigm shift across domains where numerous downstream models are directly built upon FMs which are pre-trained once but can be fine-tuned at scale. Meta-learning provides a systematic framework for learning (via pre-training) the individuality and commonality of ⟨`app, env`⟩ pairs in a systems management task, thereby demonstrating fast adaptability in fine-tuning. In contrast, simply training on a wide range of ⟨`app, env`⟩ pair leads to average or suboptimal performance while training on a narrow distribution of applications or environments results in poor generalization [60, 38]. For example, resource autoscaling policies vary with resource consumption characteristics, workload sensitivity to different resource allocations, and heterogeneous SLOs. On the other hand, training one model per application or environment leads to significant overhead [47, 54], while there is a lack of a principled and systematic approach for fine-grained clustering and training one model per cluster.

Model Size and Complexity. Preliminary results in our experiments show that a two-layer bidirectional GRU (RNN) followed by a fully connected layer can already provide 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 [55]). However, a larger model may lead to unnecessarily higher pre-training costs and inference overhead which could be detrimental to latency-sensitive online systems management tasks (e.g., job scheduling or autoscaling [39]).

Trade-offs between Generalizability and Heterogeneity. An open problem for cloud systems management is how to achieve ML model generalizability across both (1) cloud systems (i.e., applications and environments) and (2) management tasks while still allowing the model to capture the heterogeneity of the various systems in a task. FLASH is proposed as an initial FM design to facilitate adaptability across applications/environments (but ***not* across tasks**). For example, the pre-trained meta-learner from workload autoscaling may work for resource configuration search (by generating application-representative embeddings) but not for congestion control (whose management target is a traffic flow). Ultimately, we desire an FM that can learn general systems management policies while still memorizing both task-specific details and application characteristics. However, such a task-aware FM requires both a homogeneous task specification structure and unified data representation modeling that can be generalized across systems management tasks, similar to a unified next-token prediction task across different NLP-related tasks. In addition, focusing more on generalizability may introduce unavoidable intrinsic model bias in downstream systems management tasks while focusing more on heterogeneity may hinder learning an optimal shared model.

Risk of Homogenization and Bias. As the same shared models are repeatedly reused as the basis for many applications and tasks (as described in §2.2), we enjoy the benefits of fast adaptation and training cost amortization. However, centralization also means that these shared models are singular points of failure that can radiate harm (e.g., security risks or biases) to downstream applications/tasks at scale. It is well known that FMs have the potential to amplify existing biases present in the pre-training dataset [67, 51]. For instance, a key consideration for cloud systems management could be application bias or cloud infrastructure bias due to diverse characteristics. Compared to task-specific models, FMs suffer more from such bias because (1) the training data is collected on a large scale which is likely to be dominated by over-represented cloud applications or regions; (2) the

huge number of learnable parameters and complex model structures make model interpretation and debiasing much more difficult; and (3) the bias of the FMs can be easily inherited (or even amplified) by all downstream adapted models. This indicates a pressing need for designing proper debiasing frameworks for FMs in the cloud systems management domain.

Relation to Large Language Models (LLMs). FMs are a strict superset of LLMs (e.g., GPT-3 [4]) in our definition as LLMs also follow the pretrain-finetune paradigm that FMs introduce. The term "LLMs" emphasizes the way in which these pre-trained models are produced (e.g., with large textual corpora trained using next-token prediction). Akin to how deep learning was popularized in computer vision (e.g., ImageNet, AlexNet, and ResNet) [24] but now extends beyond, FMs emerged in NLP with LLMs but FMs exist for many other modalities, including images, code, speech, proteins, and molecules. However, although LLMs are quite good at information processing or summarization, they may not be suitable for decision-making tasks or prediction tasks in cloud systems management. In FLASH, on the other hand, the original ML/RL agents (i.e., base-learner) are kept because those base-learner models have been proven to be successful in handling each downstream task (e.g., an RL framework is good at workload autoscaling [40, 59]). The pretrain-finetune paradigm of FLASH additionally benefits the base-learners by providing generalizability and fast model adaptability.

Conclusion. In this paper, we discussed the promise and challenges of developing FMs for cloud systems management tasks. We proposed an initial design of FM with FLASH and demonstrated its usage in two critical systems management tasks, namely, resource configuration search and workload autoscaling. Preliminary results show that FLASH reduces the performance degradation of the ML/RL agents by 52.3–90.5% (with no adaptation) while reducing the retraining time by a factor of 5.5× (for adaptation). We concluded this work by discussing some potential risks and challenges that must be considered when developing such general-purpose models for cloud systems management, which we believe will foster future research in this domain.

Acknowledgments

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

References

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

[2] R. Bianchini, M. Fontoura, E. Cortez, A. Bonde, A. Muzio, A.-M. Constantin, T. Moscibroda, G. Magalhaes, G. Bablani, and M. Russinovich. Toward ML-centric cloud platforms. *Communications of the ACM*, 63(2):50–59, jan 2020. ISSN 0001-0782. doi: 10.1145/3364684. URL https://doi.org/10.1145/3364684.

[3] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[4] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS 2020)*, pages 1877–1901, Red Hook, NY, USA, 2020. Curran Associates Inc.

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

[6] Z. Chen and D. Marculescu. 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)*, pages 1521–1526, 2015.

[7] K. W. Church, Z. Chen, and Y. Ma. Emerging trends: A gentle introduction to fine-tuning. *Natural Language Engineering*, 27(6):763–778, 2021.

[8] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. 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)*, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132772. URL https://doi.org/10.1145/3132747.3132772.

[9] Y. Dang, Q. Lin, and P. Huang. AIOps: Real-world challenges and research innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 4–5. IEEE, 2019.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[11] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference (Middleware 2021)*, page 248–259, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385343. doi: 10.1145/3464298.3493398. URL https://doi.org/10.1145/3464298.3493398.

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

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

[14] J. Gao. Machine learning applications for data center optimization, 2014.

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

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

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

[18] F. M. M. u. Islam and M. Lin. 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.

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

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

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

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

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

[24] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[25] X. Li, F. Tang, J. Liu, L. T. Yang, L. Fu, and L. Chen. AUTO: Adaptive congestion control based on multi-objective reinforcement learning for the satellite-ground integrated network. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC 2021)*, pages 611–624. USENIX Association, 2021.

[26] C.-J. M. Liang, H. Xue, M. Yang, L. Zhou, L. Zhu, Z. L. Li, Z. Wang, Q. Chen, Q. Zhang, C. Liu, and W. Dai. 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.

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

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

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

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

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

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

[33] G. Mariani, A. Anghel, R. Jongerius, and G. Dittmann. Predicting cloud performance for hpc applications before deployment. *Future Generation Computer Systems*, 87:618–628, 2018. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2017.10.048. URL https://www.sciencedirect.com/science/article/pii/S0167739X17312542.

[34] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NeurIPS 2013)*, pages 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc.

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

[36] A. Nichol, J. Achiam, and J. Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

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

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

[39] H. Qiu, S. Jha, S. S. Banerjee, A. Patke, C. Wang, F. Hubertus, Z. T. Kalbarczyk, and R. K. Iyer. Is function-as-a-service a good fit for latency-critical services? In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, WoSC '21, page 1–8, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391726. doi: 10.1145/3493651.3493666. URL https://doi.org/10.1145/3493651.3493666.

[40] H. Qiu, W. Mao, A. Patke, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer. SIMPPO: A scalable and incremental online learning framework for serverless resource management. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC 2022)*, pages 306–322, New York, NY, USA, 2022. Association for Computing Machinery.

[41] H. Qiu, W. Mao, A. Patke, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer. Reinforcement learning for resource management in multi-tenant serverless platforms. In *Proceedings of the 2nd European Workshop on Machine Learning and Systems*, EuroMLSys '22, page 20–28, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392549.

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

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

[44] A. A. Rusu, D. Rao, J. Sygnowski, O. Vinyals, R. Pascanu, S. Osindero, and R. Hadsell. 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.

[45] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap. Meta-learning with memory-augmented neural networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML 2015)*, pages 1842–1850. JMLR.org, 2016.

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

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

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

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

[50] H. Tian, X. Liao, C. Zeng, J. Zhang, and K. Chen. 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)*, page 261–275, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450395083. doi: 10.1145/3555050.3569125. URL https://doi.org/10.1145/3555050.3569125.

[51] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

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

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

[54] M. Vartak and S. Madden. 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.

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

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

[57] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Survey*, 53(3), jun 2020. ISSN 0360-0300. doi: 10.1145/3386252. URL https://doi.org/10.1145/3386252.

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

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

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

[61] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. 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)*, page 452–465, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350280. doi: 10.1145/3127479.3131614. URL https://doi.org/10.1145/3127479.3131614.

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

[63] A. Yeganeh-Khaksar, M. Ansari, S. Safari, S. Yari-Karin, and A. Ejlali. 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.

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

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

[66] R. Zhang, X. Hu, B. Li, S. Huang, H. Deng, Y. Qiao, P. Gao, and H. Li. Prompt, generate, then cache: Cascade of foundation models makes strong few-shot learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2023)*, pages 15211–15222, 2023.

[67] Y. Zhang, F. Feng, X. He, T. Wei, C. Song, G. Ling, and Y. Zhang. Causal intervention for leveraging popularity bias in recommendation. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, page 11–20, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380379. doi: 10.1145/3404835.3462875. URL https://doi.org/10.1145/3404835.3462875.

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

[69] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou. 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)*, page 167–181, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446693. URL https://doi.org/10.1145/3445814.3446693.

A Details of Case Study on Sizeless

Sizeless Model. Sizeless [11] uses a fully connected neural network as the predictor for the regression task of resource configuration search. The label and features used in the Sizeless model are shown in Table 2, which are consistent with the original paper [11]. Sizeless predicts the serverless function execution time given a memory size allocated to the function without using CPU allocation because CPU allocation is proportional to the allocated memory in AWS Lambda. After a grid search to tune the hyperparameters of the model (as shown in Table 3), the final model uses the Adam optimizer, a MAPE loss function, 200 epochs, an L2 regularization of 10^{-2}, and four layers.

Table 2: Features and labels in Sizeless (§2.1).

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

Table 3: 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

Applications and Sizeless Dataset. We adopted the 16 representative production cloud workloads selected in Sizeless [11] based on a survey of 89 industry use cases of serverless computing applications [12]. 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 is collected by running 2000 synthetic AWS Lambda applications generated by combining the 16 selected representative production application segments based on random sampling with replacement from the segment pool. Each segment represents the smallest granularity of common workloads in cloud datacenters. In addition, each segment has to be associated with its own inputs to simplify load generation (e.g., the image manipulation workloads come with random images). The generator also comes with setup and tear-down scripts for all external services each segment uses (e.g., databases or messaging queues) on AWS (for better integration with AWS Lambda).

The original Sizeless dataset includes measurements on the execution time and resource consumption metrics (see [11] 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.

- **OpenWhisk Dataset.** We implemented a synthetic application generator based on the open-sourced generator from Sizeless. 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 described in Appendix B). Following the same dataset collection methodology as Sizeless [11], we deployed all applications on a 50-VM OpenWhisk cluster setup on IBM Cloud. In addition to function container memory size, we also consider CPU allocation (i.e., `cpu.shares` used in OpenWhisk) as another resource configuration. Collected metrics remain the same as the Sizeless dataset.
- **CloudBandit Dataset.** The CloudBandit dataset [23] 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 cloud computing infrastructures.

B Details of Case Study on FIRM

FIRM Model. In the task of multi-dimensional workload autoscaling, FIRM [38] uses an actor-critic RL algorithm, DDPG [27]. The RL agent monitors the system- and application-specific measurements and learns how to scale the allocated resources vertically and horizontally. Table 4 shows the model's state and action spaces. Table 5 shows the model hyperparameters.

The goal of the RL agent 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, $SP = 1$. 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).

Table 4: RL formulation in FIRM (§2.1).

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 5: 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})

Applications and Traces. As mentioned in Appendix A, we generated 1000 synthetic applications (deployable on both OpenWhisk and Kubernetes) using the selected 16 representative production cloud serverless workloads as application segments. We reused 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 used real-world datacenter traces [68] released by Microsoft Azure, collected over two weeks in 2021. Next, we deployed the selected workloads as Deployments in a five-node Kubernetes cluster on IBM Cloud Virtual Private Cloud (VPC) 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.

Application Updates/Patches. We introduced, in total, seven scenarios to investigate model performance degradation when facing application patches, service payload size changes, and load pattern variations. The several scenarios that we consider in this case study are: (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 was changed from [128 KB, 256 KB, 384 KB] to [512 KB, 768 KB, 1024 KB]. (2) For database services, the size of the table being scanned was changed from 1024 items to 10240 items. (3) For floating-point number calculation, the number of operations was changed from 10^8 to 20^8. (4) For image manipulations, the dimension was changed from 40×40 to 160×160. (5) For text processing, the JSON file size was changed from [250 B, 500 B, 1 KB] to [2 KB, 3 KB, 5 KB]. (6) For ML model serving, we changed the matrix multiplication dimension from 50 to 150. (7) For load pattern changes, we divided the Azure workload traces into two parts, one half with a higher daily load ($> 10^5$ per day) and the other half with a lower daily load ($\leq 10^5$ per day).

C Meta-Learner Design

As the core component of FLASH, the meta-learner is designed to explicitly model the *individuality* and the *commonality* of applications and environments in an ML-managed system. Specifically, FLASH learns a shared model/policy to characterize the task commonality (across applications and environments) and simultaneously trains a meta-learner to abstract the individuality to adapt the shared policy to specific $\langle$`app, env`$\rangle$ pairs (in the base-learner). 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, $\langle$`app, env`$\rangle$ pairs with similar characteristics are projected to neighboring locations, while those with quite

different characteristics are projected to locations far from each other. As mentioned in §2.2, the meta-learner samples labeled data points (in SL) or RL trajectories and generates an embedding that accurately represents the application running in the environment. The embedding is then fed to the base-learner (as part of its feature vector or state vector) to adapt (fine-tune) its model or policy by differentiating heterogeneous applications and cloud environment changes.

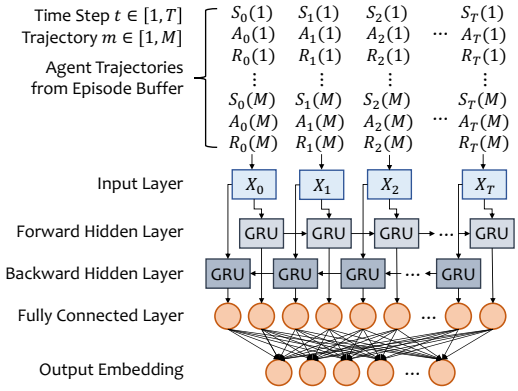


Figure 7: Neural network architecture of FLASH's RNN-based meta-learner for embedding generation. *Each variable X_t in the input layer corresponds to a vector of indexed training data samples. In supervised learning (not shown in the figure), $X_t = [x_t, y_t]$ where x_t is the feature vector and y_t is the label. In RL, $X_t = [s_t(i), a_t(i), r_t(i)], i \in \{1..M\}$ 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 in an T-step trajectory, respectively.*

Network Architecture. As shown in Fig. 7, the network architecture of FLASH's meta-learner consists of an input layer, a recurrent neural network (RNN) layer, and a fully connected neural network (FCNN) layer (i.e., the embedding layer).

- *Input Layer*: The input layer selects what kind of information FLASH can use to learn such a low-dimensional space to generate embeddings. Based on the insight that the datasets used for base-learner model training already contain spatial and temporal characteristics that the managed system manifests, labeled data points (in SL) and trajectories (in RL) are used as inputs to the meta-learner. However, for RL, simply using all episodes is computationally intensive in practice. As the goal is to learn a discriminative embedding to characterize the environment, the episodes with low rewards are unhelpful or even harmful. Intuitively, those lower-reward trajectories are generated with a random policy or a poorly trained policy, so they are not representative of the workloads. Therefore, we chose the top M trajectories that have resulted in the highest rewards.
- *RNN Layer*: We use a bidirectional GRU (a special class of RNN) [48, 15, 46] 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 memory-less 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 but not the future. Bidirectional RNNs, on the other hand, duplicate the RNN processing chain so that the inputs are processed in both forward and backward orders to enable looking into future contexts as well. GRU, compared to vanilla RNNs, combines an update gate and a reset gate to enable retaining relevant information from long sequences and discarding irrelevant or outdated information.
- *Embedding (FCNN) layer*: The output from the bidirectional RNN of the meta-learner is fed to a fully connected neural network layer to generate an embedding (i.e., a vector of fixed size) that is used to fingerprint/represent the $\langle$`app, env`$\rangle$ pair with which the base-learner is dealing with. As shown in Fig. 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.

Embedding Generation with a Bi-directional RNN. FLASH uses a bidirectional RNN [46] that maintains a high-dimensional hidden state with nonlinear dynamics to acquire, process, and memorize knowledge about the current $\langle$`app, env`$\rangle$ pair. An RNN is a type of neural network that is specialized

for processing a sequence of data $X_1, X_2, ..., X_T$ 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 [37] to the input sequences. Two unidirectional RNN hidden layers are chained together in opposite directions and acting on the same input (as shown in Fig. 7). For the forward RNN hidden layer, the first input is X_1 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_1. 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 $O_M = \frac{1}{M}\sum_{m=1}^{M} H(I_m)$ where $H(.)$ is the encoder (i.e., the RNN layer) which maps the input sequence I_m to a low-dimension vector.

Table 6: 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

We do not explicitly use the popular memory augmentation technique [45] for the meta learner as the features of our application workloads are not as high-dimensional as those of computer vision tasks [45] and the RNN hidden states suffice to provide good representations in our experiments. We also leave the usage of more advanced sequence models such as long short-term memory (LSTM) [53, 16] and attention-based techniques (e.g., transformers [55]) to our future research.

The output from the bidirectional RNN layer (i.e., O_M) is fed to a fully connected neural network (FCNN) layer to generate an embedding that is used to fingerprint/represent the $\langle$`application, environment`$\rangle$ pair with which the base learner is dealing with. 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 SL feature vector or RL state vector at each time step. We implemented FLASH's meta-learner with PyTorch, and the hyperparameters are shown in Table 6.