CLOUD SYSTEMS MANAGEMENT WITH EFFICIENT AND ROBUST ONLINE LEARNING

BY

HAORAN QIU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Doctoral Committee:

Professor Ravishankar K. Iyer, Chair
Professor Tamer Başar
Professor Klara Nahrstedt
Professor Indranil Gupta
Professor Onur Mutlu, ETH Zurich

ABSTRACT

Large-scale cloud computing systems rely heavily on decision-making algorithms for critical system management tasks such as resource allocation, job scheduling, and power management. Manually crafted heuristics for these algorithms become increasingly untenable given the complexity and heterogeneity of modern cloud environments because the intricate interactions across diverse workloads, hardware platforms, and operating conditions make it exceedingly difficult to devise fixed heuristics that work well across all scenarios. Although machine learning techniques have been proposed to automatically learn optimized system management policies, existing approaches face practical limitations and lack the robustness necessary for production-grade cloud systems.

This dissertation pioneers a novel abstraction-driven paradigm of efficient and robust online learning to fundamentally transform cloud systems management at scale. We develop a general framework that leverages deep reinforcement learning with system domain knowledge at its core to discover optimized management policies by continuously exploring and refining them through *in situ* interactions with cloud environments. To practically and robustly apply learning in cloud systems at scale, we further design two key abstractions: (1) A *virtual agent* abstraction that coordinates the distributed learned policies to resolve multi-agent interferences and stably converge on system-wide objectives, and (2) A *meta learner* abstraction that extracts generalizable policy embeddings that can rapidly adapt across the breadth of heterogeneous cloud applications and platforms. This abstraction-driven approach provides practical and extensible support for a wide range of online learning algorithms and diverse systems management tasks.

Instantiated in systems like FIRM [1] (for microservices), SIMPPO [2] (for serverless computing), and μ-Serve [3] (for deep learning model serving), our innovative framework delivers order-of-magnitude improvements in resource efficiency, performance isolation, power optimization, and generalization compared to traditional heuristic-driven approaches. More profoundly, it establishes the foundations for practical and robust autonomous cloud systems management. Our contributions span the full stack, from mathematical models and optimizations to system design, implementation, and deployment.

To my beloved family, for their dedicated love and support.

ACKNOWLEDGMENTS

This incredible journey would not have been possible without the support and contributions of many cherished individuals. I am deeply grateful for the brilliant minds and warm hearts that made my PhD experience at the University of Illinois, Urbana-Champaign (UIUC) truly remarkable.

First and foremost, I want to express my sincere gratitude to my PhD advisor, Prof. Ravishankar K. Iyer. Since I joined UIUC in 2019, he has been a wonderful source of advice, guidance, inspiration, and support. I am also grateful to Prof. Zbigniew T. Kalbarczyk and the other members of my committee, Prof. Tamer Başar, Prof. Klara Nahrstedt, Prof. Indranil Gupta, and Prof. Onur Mutlu, for their insightful comments and suggestions that improve this dissertation. Special thanks to Prof. Tianyin Xu for his mentoring in my academic career journey and his dedication to CS 523, which stands out as one of the most enlightening and intellectually stimulating courses I have taken at UIUC. Their mentorship played a pivotal role in shaping my research and personal growth.

I am also indebted to my fantastic collaborators, Saurabh Jha, Subho S. Banerjee, Weichao Mao, Archit Patke, Chen Wang, Hubertus Franke, Alaa Youssef, Qiaobin Fu, Pulkit Misra, Yawen Wang, and Phitchaya Mangpo Phothilimthana, for the countless discussions, suggestions, and detailed guidance on my research. Their expertise and contributions significantly enriched this work. My internships and visiting experience at IBM Research, Google, and Microsoft Research have inspired me to think about the real problems in large-scale cloud datacenters, which had a great impact on this dissertation.

My heartfelt thanks go to the fellow members of the DEPEND Research Group, both past and present, for creating an intellectually stimulating, friendly, and supportive environment: Subho Banerjee, Saurabh Jha, Archit Patke, Krishnakant Saboo, Chang Hu, Phuong Cao, Yoga Varatharajah, Key-whan Chung, Anirudh Choudhary, Mosbah Aouad, Haotian Chen, Yurui Cao, Shengkun Cui, Harshitha Sreejith, Pranav Dorbala, Jack Chen, and Linghao Zhang. They are all brilliant, and they have been sharing their wisdom and valuable advice with me, which has inspired my research in many ways. And to our DEPEND past and current program managers, Kathleen Atchley, Shannon Weick, Jeni Summers, and Heidi Leerkamp, thank you for being tremendously caring for your continuous support.

In addition to my friends in DEPEND, I am deeply grateful for the friendships that made my PhD journey full of fun and laughter, even during the challenges posed by the COVID-19 pandemic. Those include my roommates, Jiaqi Guan, Weichao Mao, and Dawei Sun;

friends from other labs, Yifan Zhu, Mengchao Zhang, Gao Tang, Dongqi Fu, Wale Salaudeen, Yongzhou Chen, Beitong Tian, and Xiangyuan Zhang; and my basketball buddies, Yifei Zong, Mingrui Xu, Zhengyuan Xue, Yida Li, Ziyi Li, Qian Ai, Yufan Zhang, Yin Liu, Kaiyuan Li, and Yubo Zou.

Last but not least, I would like to express my deepest gratitude to my parents and my wife for their unconditional love, unwavering support, and continuous encouragement. Their belief in me has been a constant source of strength, and I could not have achieved this milestone without them by my side.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 OVERVIEW

Amid the rapid growth of cloud adoption, efficiency has emerged as the paramount objective for achieving sustainable growth in cloud datacenters. The fundamental principle of cloud computing is to virtualize applications to run on shared hardware infrastructure. However, the rapid evolution of cloud application paradigms, such as microservices and serverless computing, coupled with the rise of machine learning workloads, has fueled increasingly heterogeneous and elastic resource demands. Concurrently, the hardware underpinning cloud infrastructure is facing decelerating scaling laws, including Moore's Law, leading to diminishing returns in performance, memory or storage costs, and power usage effectiveness.

The mounting tensions between ever-increasing application demands and the slowdown of hardware scaling laws necessitate prioritizing efficiency as the cornerstone of next-generation cloud computing. Efficiency is multi-faceted – it encompasses (1) *performance efficiency* to maximize application performance (e.g., to minimize application runtime) on fixed hardware resources, (2) *power efficiency* to minimize energy consumption through techniques like processor frequency scaling, and (3) *resource efficiency* to utilize minimal cluster resources while meeting application service-level objectives (SLOs). Optimizing for any single dimension is insufficient; a holistic and balanced approach is imperative.

However, the path to attaining such holistic efficiency is obstructed by the growing complexity and scale of cloud systems. These systems have to manage all physical and virtualized hardware and software resources while orchestrating critical management functions like scheduling, health monitoring, failure diagnosis and recovery, congestion control, and power management, adapting to the continuous evolution of cloud application computing paradigms. Therein lies the fundamental challenge – *how can we design and operationally manage these large-scale, heterogeneous, and complex cloud systems to continuously and effectively balance and meet the multi-faceted efficiency requirements?*

Over the years, researchers have turned to incorporating machine learning (ML) methods in the design and management of cloud systems. Unlike conventional heuristic-based approaches or performance modeling, ML techniques can better scale and adapt to the diverse variations across machine configurations, workloads, and deployment environments without recurring human-expert effort. These data-driven ML approaches have shown promising improvements across many areas of cloud systems – job scheduling, resource management, database indexing and query optimization, network control, configuration optimization, and more. The abundance of training data available in cloud systems provides a valuable feedback loop to train and continuously refine ML models for optimizing end-to-end objectives such as cluster resource utilization or application latency SLOs.

However, developing *robust* ML models that can reliably optimize cloud efficiency across all scenarios remains an open challenge, especially in production cloud systems. Even carefully designed heuristics and human operators can make mistakes, leading to cluster-wide degradations from incorrect resource scaling, power control, or misconfigurations. Our study (through collaboration with IBM) further revealed how ML models can fail or underperform due to violated modeling assumptions in production cloud environments.

This refines the core research problem to: "*How can we achieve truly efficient and robust learning for cloud systems?*", which is critical for the practical adoption of ML techniques in production cloud systems. In this dissertation, we take a step back and ask what is the most natural way for ML to be integrated with complex cloud systems. Rather than designing and tuning ad hoc ML models for each system management problem, we seek to design a general framework that empowers cloud systems to automatically learn and optimize for efficiency objectives. Our proposed approach shifts the burden away from system operators having to design specialized heuristics or worry about adapting ML models to various corner cases to avoid robustness failures. Instead, the system operators architect the framework with data collection, experimentation, and learning objectives.

With the proposed approach, the ML models in the cloud system then become *learning agents*, continuously improving cloud efficiency by exploring the operating state space guided by the learning framework while inheriting extensible robustness capabilities (to deal with external disturbances like data quality issues, cloud environment heterogeneity, and interferences from other agents). The aim is to pave the way towards an era of self-driving, self-optimizing cloud computing infrastructure that can harmonize multi-dimensional efficiency targets in a fundamentally more autonomous and robust manner.

1.2 GENERAL APPROACH: EFFICIENT AND ROBUST ONLINE LEARNING

The broad vision of this dissertation is to bridge the gap between advanced ML techniques and their robust integration within cloud systems via data-driven optimizations. In particular, we aim to build efficient and robust learning-based *system policies* (i.e., control algorithms) that make decisions during the operations of a cloud datacenter. We focus on cloud system management tasks at various system stacks where the management *agents* in each task continue to learn and improve those system policies. For instance, in a workload autoscaling task, an autoscaler (agent) determines the CPU/memory allocation of the

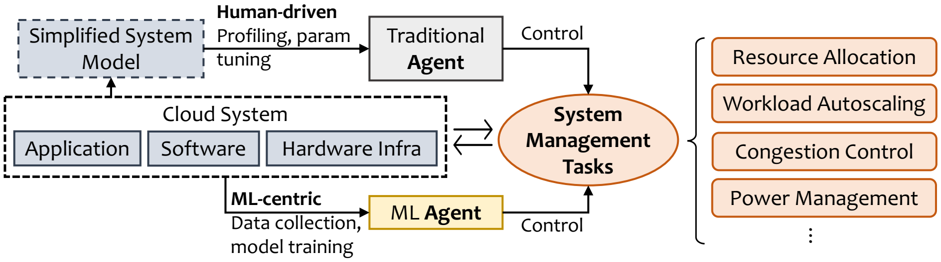


Figure 1.1: An ML-centric cloud system management with a mix of non-ML- and ML-based system management agents.

deployed containers or virtual machines, reacting to workload demand variations.

In practice, traditional system management agents in cloud datacenters are based on handcrafted heuristics or static policies. For example, Kubernetes horizontal autoscalers make decisions to adjust the number of container replicas for a running application job based on fine-tuned CPU utilization thresholds. As shown in Fig. 1.1, such heuristics design flow is human-driven and involves cloud systems modeling, profiling, and parameter tuning. Traditional heuristics-based agents have been successful in static settings where we can model the system quite accurately. However, variations across cloud hardware infrastructure, application workloads, or deployment environments can make heuristic generation repetitive and costly. In contrast, ML-based approaches propose to replace recurring human-expert-driven engineering efforts with ML model data collection and model training (as shown at the bottom of Fig. 1.1) with the following benefits that traditional agents struggle to offer:

(1) Learning optimal policies for a specific environment by directly optimizing for the actual operating conditions instead of relying on inaccurate system models.

(2) Optimizing for end-to-end system objectives directly (e.g., application latency SLOs) without prior knowledge of how low-level system metrics impact the objective.

(3) Handling hard-to-model system dynamics (e.g., performance interference through last-level cache contention) with the use of general-purpose and powerful function approximators such as deep neural networks.

Our work continues along the lines of ML-centric cloud systems management in replacing human-driven engineering efforts with automation. While the classical paradigm of building such ML agents has demonstrated success in limited homogeneous settings, the use of ML in systems today still falls short (especially in production cloud systems) as we currently lack design techniques and a systematic framework for achieving continuous cloud efficiency optimization across heterogeneous cloud environments and application computing requirements.

To bridge this gap, we augment the computer system with data-driven learning techniques and provide a general framework with both system and algorithmic support to enable efficient and robust online learning. This framework helps overcome three fundamental challenges in ML-centric cloud systems management.

Complexity and Dynamics. Modern hardware and software stacks expose a diverse array of configurable parameters whose intricate interactions can have complex and unpredictable impacts on performance and reliability. Therefore, cloud systems must efficiently adapt to constantly evolving operating conditions, fluctuating workload patterns, and shifting user requirements. Given a well-defined system management task, our approach combines deep learning techniques, capable of modeling such complexity, with domain-driven architectural model decomposition designed explicitly to address this challenge and handle system dynamics. The produced ML models (used by the agents to perform the given system management task) are efficient in training and interpretable for debugging.

A key enabler is the reinforcement learning (RL) paradigm. The agent starts with no prior knowledge about their assigned control tasks. They learn optimal policies by receiving reward signals that evaluate how well they are performing on the task objective. Through repeated interactions with the system environment, the agents accrue experience to progressively improve their decision-making over time. Since in the cloud systems domain, management decisions tend to be recurrent in nature, the agent can readily collect large volumes of training data to train the models.

Cloud Multi-tenancy. With multiple agents deployed to control different systems tasks and manage applications from different tenants with varying characteristics and requirements, our framework enables cross-stack policy optimization by having the agents share a common reward signal. This joint optimization capability is a key advantage over traditional heuristic-based approaches that operate in isolated silos. The data-driven nature allows the system to dynamically adjust to optimal operating states by continuously ingesting and adapting to the contextual telemetry data collected across all system layers - capturing the precise deployment environment, workload patterns, and application requirements. However, the multi-agent setting also introduces new challenges around conflicts between agents' policies as well as the non-stationarity caused by agents' simultaneous learning and interaction in the shared cloud environment.

To address this challenge, our framework employs mean-field theory techniques to model and coordinate the agent population at scale. This allows the agents to coexist stably, manage perturbations from concurrent learning, and ultimately converge to a coherent system-wide optimum. The mean-field approach provides the crucial scalability needed to orchestrate a large number of control agents spanning the cloud systems stack.

Cloud Heterogeneity. Another core challenge is the heterogeneity intrinsic to cloud environments spanning diverse applications and infrastructure. As the learned ML model is specialized to the environment- or application-specific characteristics (because the RL policy is trained to maximize the cumulative rewards), directly applying such ML models as the control agents incurs uncertain performance degradation and significant adaptation overhead. To mitigate this, our framework leverages meta-learning to produce general embedding representations that help rapidly specialize to novel applications and environments with limited fine-tuning costs.

At its core, this dissertation takes a general approach towards realizing self-driving cloud datacenter management that combines deep learning with systems principles. The inherent complexity and dynamics are tamed by modularizing control into distinct tasks amenable to deep reinforcement learning so that the learning agents can continuously adapt and refine their policies through constant environment interaction. Potential conflicts between the distributed agent policies are resolved through mean-field population modeling, allowing the agents to stably co-exist and converge. Furthermore, the framework leverages meta-learning to bridge the cloud heterogeneity gap, facilitating efficient adaptation of learned models across diverse applications and cloud environments. This general design establishes the foundations for fundamentally autonomous cloud datacenter management.

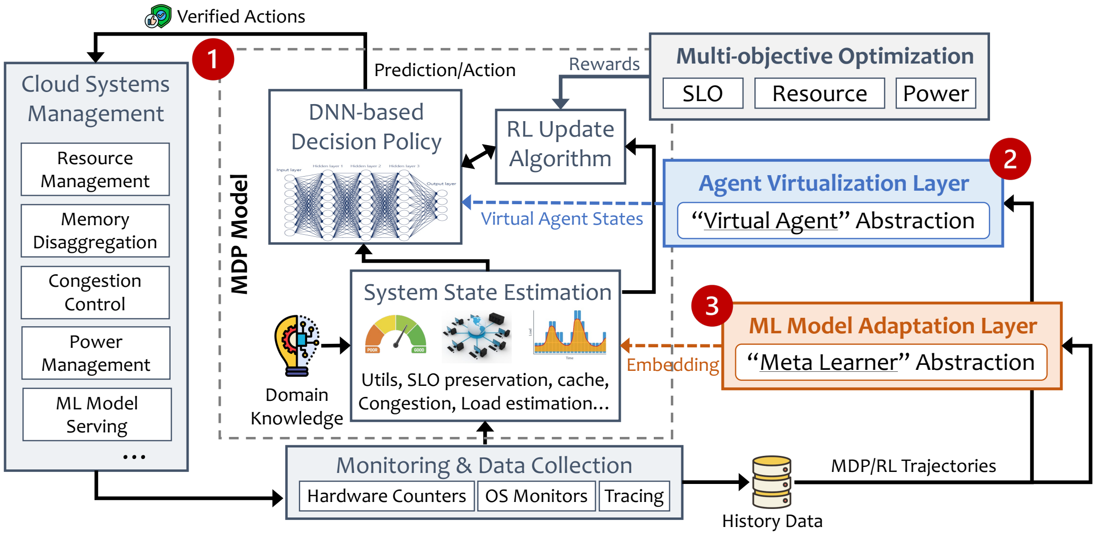


Figure 1.2: Overview of the proposed system architecture for efficient and robust online learning in cloud systems.

1.3 SUMMARY OF CONTRIBUTIONS: METHODS AND SYSTEMS DEVELOPED

This dissertation develops a set of novel, transformative methods to control, manage, and optimize large-scale heterogeneous cloud datacenters with efficient and robust online learning. We demonstrate their uses and generalizability in several representative tasks, including resource management, processor frequency scaling, and deep learning model serving, but the methods are generalizable to a wider range of cloud systems management tasks. Fig. 1.2 provides an overview of the proposed methods that leverages the rich monitoring and telemetry data available across the stack to model different aspects of the heterogeneous system behavior using Markov decision process (MDP) formulations augmented with domain knowledge. At its core, the dissertation tackles the following key research problems.

1.3.1 FIRM: Learning Cloud Microservices Resource Management Policies

We started by looking at resource management, a core task in cloud datacenter management. As containerization transforms the datacenter from *machine-oriented* to *application-oriented* [4], meeting strict cloud application SLOs (e.g., in throughput and, more critically, tail latency [5, 6, 7, 8]) poses significant challenges. In addition, cloud services have recently undergone a major shift from complex monolithic designs to *microservices* that execute on shared/multi-tenant compute resources either as virtual machines (VMs) or as containers (with containers gaining significant popularity of late). These microservices must handle diverse load characteristics while efficiently multiplexing shared resources to maintain SLOs on application metrics such as end-to-end latency. In this dissertation, we present FIRM (labeled as ❶ in Fig. 1.2), a multilevel ML-based resource management framework to manage shared resources among microservices at finer granularity to reduce resource contention and thus increase performance isolation and resource utilization. We provide an open-source implementation of FIRM for the Kubernetes container orchestration system. FIRM reduces overall SLO violations by up to 16× compared with Kubernetes autoscaling while reducing the overall requested CPU by as much as 62%.

1.3.2 SIMPPO: Dealing with Cloud Multi-tenancy

The challenge is to reconcile cloud system management decisions from multiple ML agents in a shared cloud platform. The standard assumption for any learning-based approach is *environment stationarity* [9, 10, 11], which means that the cloud environment is affected only by the agent interacting with it. In contrast, the computing platforms (e.g., container orchestration platforms and serverless computing platforms) in any cloud datacenter are multi-tenant, and application workloads from all customers sharing the compute resources in a cluster. Multi-tenancy makes the environment *non-stationary* from each agent's own perspective when all agents are jointly being trained or used for inference. Single-agent solutions, where one agent manages one or more applications in one specific task, paying no attention to any other agents, fail to converge. On the other hand, centralized multi-agent solutions [12] cannot scale beyond a few agents in practice because of their exponential dependence on the number of agents. This dissertation presents SIMPPO (labeled as ❷ in Fig. 1.2), a scalable multi-agent framework that addresses the challenge in two ways: (1) a *virtual agent* abstraction that consists of the environment and all the other agents so that the many-agent problem is converted to a two-agent problem, and (2) an auxiliary global state modeling of the collective behavior of the other agents (i.e., the virtual agent) with mean-field theory. SIMPPO is a general framework that can support a variety of ML agents that employ online learning algorithms. Experiments in the task of serverless resource management demonstrate that SIMPPO enables all agents' policies to converge when they are jointly trained and achieves 4.5× improvement in online policy-serving performance compared to single-agent solutions (e.g., FIRM) in multi-tenant cases.

1.3.3 AWARE & FLASH: Dealing with Cloud Heterogeneity

The challenge is to quickly adapt the trained model to heterogeneous cloud applications and environments that could be unseen during training, which is a critical problem in making ML practical in production systems. For example, in resource management, a learned policy is application-specific and environment-specific. Retraining is needed to adapt to a new workload or underlying infrastructure in heterogeneous and dynamically evolving (possibly multi-cloud) datacenters [13, 14, 15, 16, 17]. FIRM's trained RL policy suffers performance degradation and requires substantial retraining (with additional data collection). In this dissertation, we present AWARE (labeled as ❸ in Fig. 1.2) that addresses cloud heterogeneity in two ways: (1) a *meta learner* abstraction that leverages meta-learning to model the RL agent as a base learner and learns to generalize and adapt to new applications and environment shifts with a novel embedding generation framework, and (2) continuous monitoring for retraining detection and trigger to achieve stable online RL policy-serving performance. AWARE adapts learned resource management policies to new workloads 5.5× faster than existing transfer-learning-based approaches. In FLASH, we extend the meta-learning approach to support both supervised learning and RL in more system management tasks: resource configuration search, CPU frequency scaling, and network congestion control.

1.3.4 μ-Serve: Learning Deep Learning Model Serving Policies

Over the past few years, increasingly capable large deep learning (DL) models have been developed for everything from recommendations to text or image generation. Pre-trained DL models make it easy for developers to build and deploy new models with lightweight fine-tuning, few-shot learning, or even prompting [18]. As a result, model serving (i.e., inference) has become an essential workload in modern cloud systems [19]. As DL workloads exhibit significantly different characteristics and requirements compared to traditional cloud applications, we aim to extend the developed learning-based framework (i.e., as shown in Fig. 1.2) to support DL model-serving workloads in cloud datacenters. The goal is to minimize power consumption while preserving SLO attainment. In particular, we address two unique challenges. First, generative model (e.g., large language models like GPTs) serving is non-deterministic because of the *autoregressive* nature[1] of generative models. Therefore, first-come-first-serve (FCFS) request scheduling policy suffers from *head-of-line blocking* issues [20], leading to less power-saving headroom without SLO violations. Second, unlike CPUs, GPUs do not support fine-grained (e.g., per-core) frequency scaling. Therefore, if the model or multiple model partitions (from different models) are placed onto a device, the frequency can only be reduced to the maximum frequency demand among all partitions. We build μ-Serve, a power-aware DL model serving framework with proxy-model-based request scheduling and fine-grained model provisioning to maximize power-saving opportunities. Thanks to μ-Serve's support of DL workloads, we can extend the developed learning-based framework for GPU frequency scaling. We evaluate μ-Serve with a diverse set of open-source DL models (including traditional non-Transformer models like CNNs, Transformers, and Transformer-based generative models) and production workloads on an 8-node 16-GPU cluster. Evaluation results show that, compared to existing state-of-the-art serving systems, μ-Serve achieves a power reduction factor of 1.9–2.6× while preserving SLO attainments.

1.4 ORGANIZATION

The remainder of the dissertation is organized as follows. First, we describe background, challenges, and related work in this area in Chapter 2. Then, Chapter 3 describes the models, design, training, and validation of a learning-based resource management policy that makes use of online telemetry data to multiplex shared resources among cloud microservices. Chapter 4 describes methods to enable joint training to convergence, mitigate agent

[1] Each model-serving request is processed iteration by iteration where the output of each iteration is used as input in the following iteration. Therefore, the total execution time is unknown before serving the request.

interferences, and thus allow ML agents to co-exist in a shared cloud platform. Chapter 5 describes methods for fast model adaptation to novel, heterogeneous cloud applications and environments. Chapter 6 describes how the proposed framework described in Chapters 3 to 5 can be extended to support deep learning workloads in cloud datacenters. Finally, in Chapter 7, we will end this dissertation by summarizing our contributions and discussing open problems and future research directions in this field.

CHAPTER 2: BACKGROUND, MOTIVATION, AND CHALLENGES

This dissertation takes a first step towards building efficient and robust cloud datacenter management systems with online learning. The implementation and deployment of such learning-augmented systems have the potential to make it much easier to deploy and manage complex cloud applications, leading to faster development, less management effort, and higher efficiency. However, one has to overcome two major challenges:

(1) *Complexity and Dynamics in Cloud Datacenters*: Modern cloud datacenter hardware, software systems, and applications exhibit heterogeneous characteristics with diverse configurable parameters that have complex interactions with performance and reliability. Worse, there can be unpredictable changes and dynamics in cloud environments, including system operating conditions and application workloads.

(2) *Robust and Practical Machine Learning in Systems*: Developing ML models that are practical for complex systems like cloud datacenters poses significant challenges. These models need to continuously deliver robust policy-serving performance in case of unexpected scenarios that can be common in cloud environments. We summarize two major sources of robustness challenges: (1) dealing with ML agent interferences due to cloud multi-tenancy, and (2) fast model adaptation to heterogeneous cloud applications and compute infrastructures.

In this chapter, we will discuss the complexity and dynamics in cloud datacenter management (Section 2.1) through a characterization study on microservices resource management (Section 2.2), motivating the use of machine learning (ML) in systems. Then we will describe the related work on ML for various system tasks (Section 2.3) and discuss the research gap addressed in this dissertation towards practical and robust learning in cloud systems management (Section 2.4).

2.1 COMPLEXITY AND DYNAMICS IN CLOUD DATACENTER MANAGEMENT

Cloud datacenters are undergoing an unprecedented evolution. On the demand side, emerging applications such as microservices [21, 22, 23, 24] and large deep learning (DL) model (e.g., large language models like GPTs [25]) training and inference require increased efficiency and scalability. In dynamically evolving (and possibly multi-cloud [17, 26]) datacenters, the underlying cloud infrastructure is becoming significantly more complex, heterogeneous, and distributed [13, 14, 15, 16]. Cloud datacenter management plays a critical role in managing and running applications on shared cloud infrastructure.

However, cloud systems have become tremendously complex during the past decade of evolution. They have to manage all the physical and virtual resources running in datacenters and take care of other management tasks such as job scheduling, server health monitoring and repairing, managing power and energy usage, and capacity planning. To continuously meet application and system service-level objectives (SLOs), one has to deal with the complexities originating from both application (e.g., computing paradigms like complex interdependencies of microservices) and cloud infrastructure (e.g., diversity of hardware/software profiles and configurations). Interactions between changes in application profiles and such hardware/software configuration parameters have become incredibly complex, and modeling those interactions with high fidelity is increasingly hard and costly. Even a modest number of distinct configuration parameters can lead to an immense search space that is impractical to exhaustively traverse during runtime execution.

In addition, dynamic cloud environments require the management system and policies to reliably adapt to unpredictable changes in the operating environment, network conditions, application workloads, and user preferences. For instance, critical execution paths in microservices do not remain static over the execution of requests but rather change dynamically based on the performance of individual service instances because of underlying shared resource contention and their sensitivity to this interference.

In the next section, we present a characterization study of cloud microservices to demonstrate the complexity and dynamics of cloud systems and to motivate the use of machine learning in resource management.

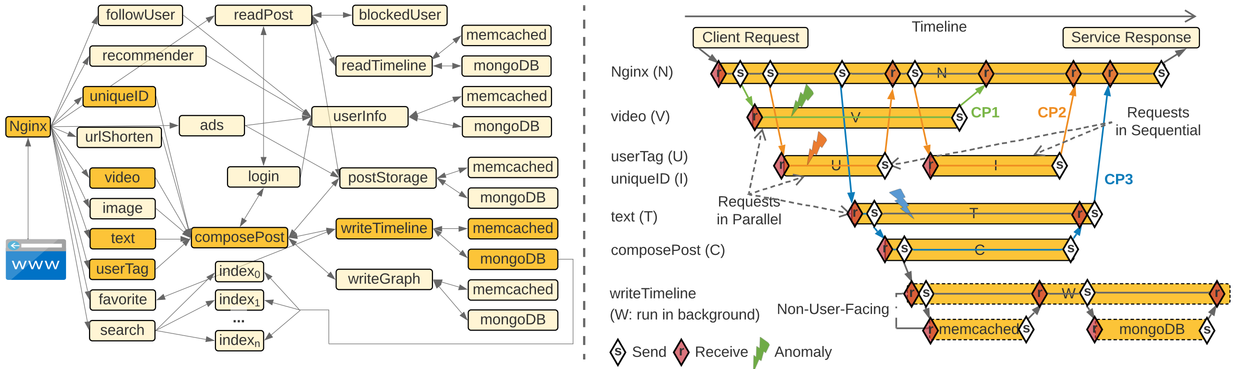
2.2 A CHARACTERIZATION STUDY ON MICROSERVICES

The advent of *microservices* has led to the development and deployment of many cloud services that are composed of “micro,” loosely coupled, intercommunicating services, instead of large, monolithic designs. This increased popularity of service-oriented architectures (SOA) of web services has been made possible by the rise of containerization [27, 28, 29, 30] and container-orchestration frameworks [31, 32, 33, 34] that enable modular, low-overhead, low-cost, elastic, and high-efficiency development and production deployment of SOA microservices [21, 22, 23, 24, 35, 36, 37, 38, 39]. A deployment of such microservices can be visualized as a *service dependency graph* or an *execution history graph*. The performance of a user request, i.e., its end-to-end latency, is determined by the *critical path* of its execution history graph. See below for the formal definitions of the terms used in this section.

Definition 2.1. A *service dependency graph* captures communication-based dependencies (the edges of the graph) between microservice instances (the vertices of the graph), such as remote procedure calls (RPCs). It tells how requests are flowing among microservices by following parent-child relationship chains. Fig. 2.1(a) shows the service dependency graph of the *Social Network* microservice benchmark [39]. Each user request traverses a subset of vertices in the graph. For example, in Fig. 2.1(a), `post-compose` requests traverse only those microservices highlighted in darker yellow.

Definition 2.2. An *execution history graph* is the space-time diagram of the distributed execution of a user request, where a vertex is one of `send_req`, `recv_req`, and `compute`, and edges represent the RPC invocations corresponding to `send_req` and `recv_req`. The graph is constructed using the global view of execution provided by distributed tracing of all involved microservices. For example, Fig. 2.1(b) shows the execution history graph for the user request in Fig. 2.1(a).

Definition 2.3. The *critical path* (CP) to a microservice m in the execution history graph of a request is the path of maximal duration that starts with the client request and ends with m [40, 41]. When we mention CP alone without the target microservice m, it means the critical path of the "Service Response" to the client (see Fig. 2.1(b)), i.e., end-to-end latency.



(a) Service Dependency Graph (b) Execution History Graph

Figure 2.1: Microservices overview: (a) Service dependency graph of *Social Network* from the DeathStarBench [39] benchmark; (b) Execution history graph of a `post-compose` request in the same microservice.

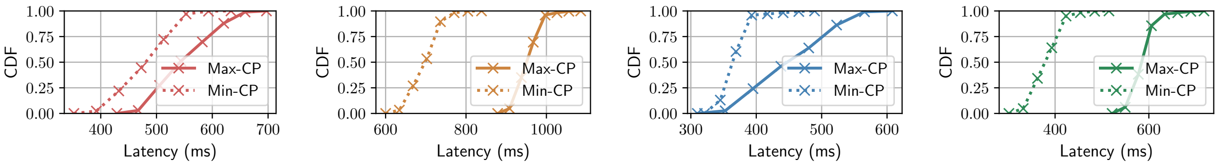
To understand SLO violation characteristics and study the relationship between runtime performance and the underlying resource contention, we have run extensive performance anomaly injection experiments on widely used microservice benchmarks (i.e. DeathStarBench [39] and Train-Ticket [42]) and collected around 2 TB of raw tracing data (over 4.1×10^7 traces). Our key insights are as follows.

Table 2.1: CP changes in Fig. 2.1(b) under performance anomaly injection.

Case	**Average Individual Latency** (ms)						**Total** (ms)
	N	V	U	I	T	C	
$<V, CP1>$	13	603	166	33	71	68	614 ± 106
$<U, CP2>$	14	237	537	39	62	89	580 ± 113
$<T, CP3>$	13	243	180	35	414	80	507 ± 75

Insight 2.1: Dynamic Behavior of CPs. In microservices, the latency of the CP limits the overall latency of a user request in a microservice. However, CPs do not remain static over the execution of requests in microservices, but rather change dynamically based on the performance of individual service instances because of underlying shared-resource contention and their sensitivity to this interference. Though other causes may also lead to CP evolution in real-time (e.g., distributed rate limiting [43], and cacheability of requested data [44]), it can still be used as an efficient manifestation of resource interference.

For example, in Fig. 2.1(b), we show the existence of three different CPs (i.e., CP1–CP3) depending on which microservice (i.e., V, U, T) encounters resource contention. We artificially create resource contention by using *performance anomaly injections.*[2] Table 2.1 lists the changes observed in the latencies of individual microservices, as well as end-to-end latency. We observe as much as 1.2–2× variation in end-to-end latency across the three CPs. Such dynamic behavior exists across all our benchmark microservices. Fig. 2.2 illustrates the latency distributions of CPs with minimum and maximum latency in each microservice benchmark, where we observe as much as 1.6× difference in median latency and 2.5× difference in 99th percentile tail latency across these CPs.



(a) Social network. **(b)** Media. **(c)** Hotel reservation. **(d)** Ticket booking.

Figure 2.2: Distributions of end-to-end latencies of different microservices in the DeathStarBench [39] and Train-Ticket [42] benchmarks. Dashed and solid lines correspond to the minimum and maximum critical path latencies on serving a request.

Recent approaches (e.g., [45, 46]) have explored static identification of CPs based on historic data (profiling) and have built heuristics (e.g., application placement, level of parallelism) to enable autoscaling to minimize CP latency. However, our experiment shows that

[2] *Performance anomaly injections* (Section 3.2.6) are used to trigger SLO violations by generating fine-grained resource contention with configurable resource types, intensity, duration, timing, and patterns, which helps with both our characterization and ML model training (Section 3.2.4).

this by itself is not sufficient. The requirement is to *adaptively capture changes in the CPs*, in addition to changing resource allocations to microservice instances on the identified CPs to mitigate tail latency spikes.

Insight 2.2: Microservices with Larger Latency Are Not Necessarily Root Causes of SLO Violations. It is important to find the microservices responsible for SLO violations to mitigate them. While it is clear that such microservices will always lie on the CP, it is less clear which individual service on the CP is the culprit. A common heuristic is to pick the one with the highest latency. However, we find that that rarely leads to the optimal solution. Consider Fig. 2.3. The left side shows the CDF of the latencies of two services (i.e., `composePost` and `text`) on the CP of the `post-compose` request in the Social Network benchmark. The `composePost` service has a higher median/mean latency while the `text` service has a higher variance. Now, although the `composePost` service contributes a larger portion of the total latency, it does not benefit from scaling (i.e., getting more resources), as it does not have resource contention. That phenomenon is shown on the right side of Fig. 2.3, which shows the end-to-end latency for the original configuration (labeled "Before") and after the two microservices were scaled from a single to two containers each (labeled "Text" and "Compose"). Hence, scaling microservices with higher variances provides better performance gain.

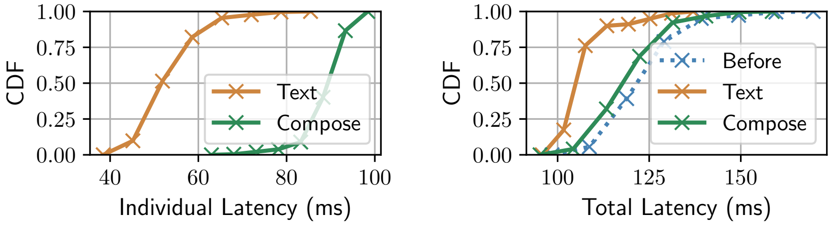


Figure 2.3: End-to-end latency improvement by scaling "highest-variance" and "highest-median" microservices.

Insight 2.3: Mitigation Policies Vary with User Load and Resource in Contention. The only way to mitigate the effects of dynamically changing CPs, which in turn cause dynamically changing latencies and tail behaviors, is to efficiently identify microservice instances on the CP that are resource-starved or contending for resources and then provide them with more of the resources. Two common ways of doing so are (1) to *scale out* by spinning up a new instance of the container on another node of the compute cluster, or (2) to *scale up* by providing more resources to the container via either explicitly partitioning resources (e.g., in the case of memory bandwidth or last-level cache) or granting more resources to an already deployed container of the microservice (e.g., in the case of CPU

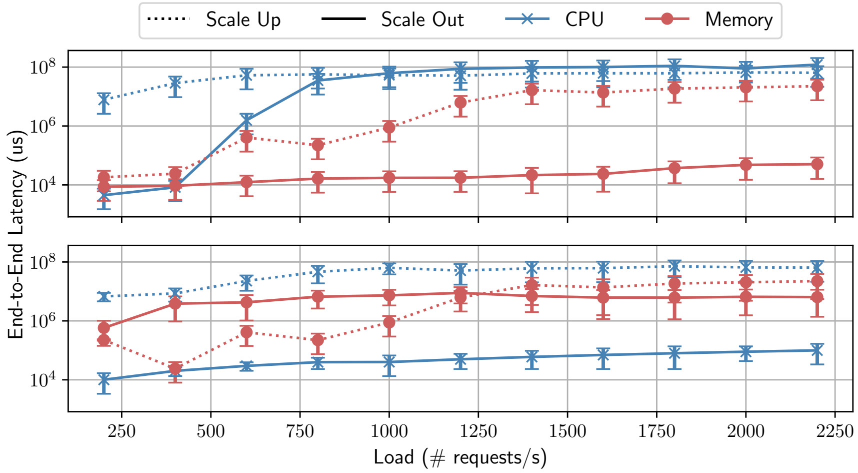


Figure 2.4: Dynamic behavior of mitigation strategies: *Social Network* (top); *Train-Ticket Booking* (bottom). Error bars show 95% confidence intervals on median latencies.

cores).

As described before, recent approaches [47, 48, 49, 50, 51, 52, 53, 54, 55]) address the problem by building static policies (e.g., AIMD for controlling resource limits [52, 53], and rule/heuristics-based scaling relying on profiling of historic data about a workload [54, 55]), or modeling performance [50, 51]. However, we found in our experiments with the four microservice benchmarks that such static policies are not well-suited for dealing with latency-critical workloads because the optimal policy must incorporate dynamic contextual information. That is, information about the type of user requests, and load (in requests per second), as well as the critical resource bottlenecks (i.e., the resource being contended for), must be jointly analyzed to make optimal decisions. For example, in Fig. 2.4 (top, for the social network application), we observe that the trade-off between scale-up and scale-out changes based not only on the user load but also on the resource type. At 500 req/s, scale-out has a better payoff (i.e., lower latency) than scale-up for both memory- and CPU-bound workloads. However, at 1500 req/s, scale-up dominates for CPU, and scale-out dominates for memory. This behavior is also application-dependent because the trade-off curve inflection points change across applications, as illustrated in Fig. 2.4 (bottom, for the train-ticket booking application).

2.3 MACHINE LEARNING FOR CLOUD SYSTEMS

Recent advances in machine learning 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 [56]. 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 [56, 57, 58]. These decisions govern how systems handle applications to satisfy user requirements in a particular runtime environment or cloud infrastructure. Various efforts have shown significant benefits in formulating production cloud system management into ML/RL-based tasks (e.g., Markov decision-making problems) including resource management [1, 59, 60, 61, 62, 63, 64], job scheduling [65, 66, 67, 68, 69], congestion control [70, 71, 72, 73, 74], and power management [16, 75, 76, 77].

Below are three examples that we use in Section 2.4 to illustrate the gap between ML advances and its practical adoption in production cloud systems.

- **Resource configuration search (RCS)** is usually modeled as a regression task [60, 61, 62, 64]. A supervised-learning-based predictor is trained to predict the application performance given the resource configurations, resource utilization, and other metrics. For example, Sizeless [64] 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[3]. PARIS [61] applies random forests to predict the 90th percentile performance metrics (e.g., latency or throughput) based on resource utilization and system-level metrics.

- **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) [1, 59]. 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) [65, 78]. For example, FIRM [1] 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 [77]. SmartOverclock [16] leverages an RL model, Q-Learning, to decide when and how much to scale the CPU core frequency.

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

While there have been successes in using ML/RL for systems management tasks as described in Section 2.3, there is still a large gap in directly applying ML/RL advances to real-world production systems due to a series of assumptions that are rarely satisfied in practice. Such a gap often prevents off-the-shelf ML/RL methods from achieving strong performance in different cloud system tasks. Here, we primarily focus on the common challenges that arise in different stages of the ML agent design pipeline across many system tasks. In this section, we look at the example of learning-based resource management as a representative system management task for concrete discussion.

Learned Resource Management Policies. RL is well-suited for learning resource re-provisioning policies, as it provides a tight feedback loop for exploring the action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules). RL automates the repetitive process of heuristics tuning and testing by enabling an *agent* to learn the optimal *policy* directly from interaction with the *environment*. It allows direct learning from the actual workload and operating conditions to understand how adjusting low-level resources affects application performance. RL consists of a *policy-training* stage and a *policy-serving* stage [79]. At the policy-training stage, the agent (using an initialized policy) starts with no knowledge about the environment and learns by reinforcement and directly interacting with the environment. At the policy-serving stage, the trained policy is used to generate an action based on the current state of the environment, and model parameters are no longer being updated.

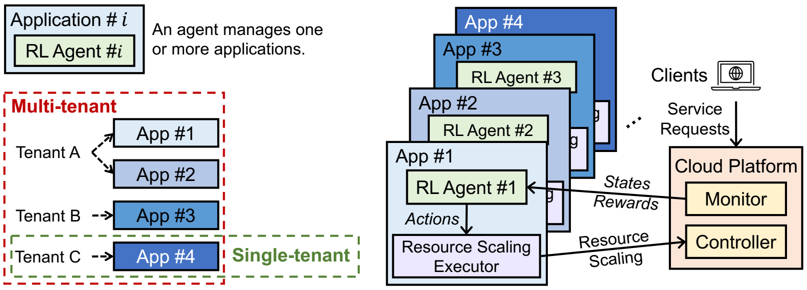
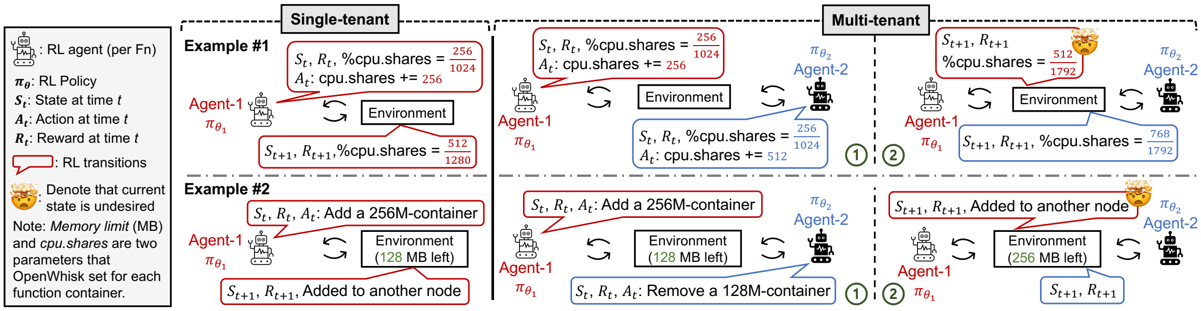


Figure 2.5: Single-agent RL solutions (each agent independently trained and unaware of each other) in single- or multi-tenant serverless environments.

Challenge of Multi-tenancy. Despite recent successes, existing RL-based solutions are all single-agent RL solutions in which one agent manages one or more applications or functions[4] (as shown in Fig. 2.5), paying no attention to any other agents. The standard

[4] RL-based approaches are all application-specific or function-specific (in the context of serverless function-as-a-service or FaaS) where one RL agent controls an application and learns to adapt to specific workloads.



: RL agent (per Fn)
π_θ: RL Policy
S_t: State at time t
A_t: Action at time t
R_t: Reward at time t
: RL transitions
: Denote that current state is undesired
Note: *Memory limit* (MB) and *cpu.shares* are two parameters that OpenWhisk set for each function container.
Single-tenant
Example #1
S_t, R_t, %cpu.shares = $\frac{256}{1024}$
A_t: cpu.shares += 256
Agent-1
π_{θ_1}
Environment
S_{t+1}, R_{t+1},%cpu.shares = $\frac{512}{1280}$
Example #2
S_t, R_t, A_t: Add a 256M-container
Agent-1
π_{θ_1}
Environment (128 MB left)
S_{t+1}, R_{t+1}, Added to another node
Multi-tenant
S_t, R_t, %cpu.shares = $\frac{256}{1024}$
A_t: cpu.shares += 256
π_{θ_2}
Agent-2
Agent-1
π_{θ_1}
Environment
S_t, R_t, %cpu.shares = $\frac{256}{1024}$
A_t: cpu.shares += 512
1
2
S_{t+1}, R_{t+1}
%cpu.shares = $\frac{512}{1792}$
π_{θ_2}
Agent-2
Agent-1
π_{θ_1}
Environment
S_{t+1}, R_{t+1}, %cpu.shares = $\frac{768}{1792}$
S_t, R_t, A_t: Add a 256M-container
π_{θ_2}
Agent-2
Agent-1
π_{θ_1}
Environment (128 MB left)
S_t, R_t, A_t: Remove a 128M-container
1
2
S_{t+1}, R_{t+1}, Added to another node
π_{θ_2}
Agent-2
Agent-1
π_{θ_1}
Environment (256 MB left)
S_{t+1}, R_{t+1}

Figure 2.6: Examples of single-agent RL failure in multi-tenant environments due to violation of the stationarity assumption. The RL environment models a systems task of multidimensional autoscaling of serverless functions. Each agent manages one application function.

assumption for an RL algorithm is *environment stationarity* [9, 10, 11], which means that the environment is affected only by the agent interacting with it. Therefore, existing single-agent RL solutions assume that the agent is in an isolated single-tenant environment (for both training and inference) that contains only the application that the agent manages. In contrast, the computing platforms (e.g., container orchestration platforms and serverless FaaS platforms) in any cloud datacenter are multi-tenant, and application workloads from all customers compete for shared resources in a cluster. Multi-tenancy makes the environment *non-stationary* from each agent's own perspective when all agents are jointly being trained. At the training stage, since the state transitions and rewards each agent gets depend on the joint actions of all agents whose policies keep changing in the learning process, each agent enters an endless cycle of adapting to other agents. At the policy-serving stage, an action might be suboptimal when applied, because the underlying environment is no longer the same as the one previously perceived for generating the action. Two examples below demonstrate such undesirable behaviors caused by environment non-stationarity.

Motivating Examples of Agent Interference. We draw two examples (as shown in Fig. 2.6) from RL episodes in our experiments where environment non-stationarity causes suboptimal agent behaviors. In both examples, each agent controls the resource management for a serverless function and is independently trained to convergence. In the first example, suppose that agent-1 makes the decision to scale up `cpu.shares` in the single-tenant case (upper left). When both agent-1 and agent-2 are present (upper right), the action from agent-2 (i.e., scaling up `cpu.shares` by 512) affects the final CPU share ratio for both agents. Our evaluation shows that the suboptimal policy results in up to 14× performance degradation compared to the single-tenant case in terms of the end-to-end p99 latency. In the second example, agent-1 wants to scale out by adding a 256-MB container, but it ends by placing the container on a new server, since the available memory capacity is only 128

MB (bottom left). Consequently, function performance will be significantly affected by the launch of a new VM and cold starts [80]. When both agents are present (bottom right), the scale-in action from agent-2 avoids involving another server. Since agents are unaware of each other, the optimal solution is missed, leading to SLO violations and inefficient utilization of resources.

We implemented a state-of-the-art single-agent RL solution, conducted training convergence analysis, and evaluated the function performance and resource utilization. We found that the single-agent RL solution converges in isolated environments and provides a sufficient baseline compared with heuristics. However, system support for many-agent RL-based resource management that provides both training convergence and comparable policy-serving performance is needed to deal with environment non-stationarity.

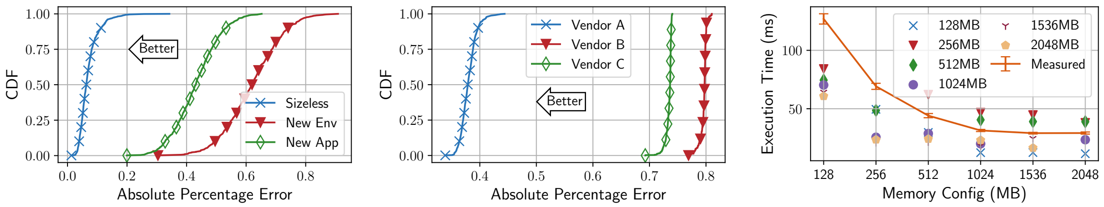
Challenge of Dynamic and Uncertain Environment. Serving an RL agent in a production cloud system also involves challenges in both policy training and serving. First, a learned RL policy is workload-specific and infrastructure-specific. Retraining is needed to adapt to a new workload or underlying infrastructure in heterogeneous and dynamically evolving (possibly multi-cloud) datacenters [13, 14, 15, 16, 17]:

- *Environment diversity and infrastructure dynamics.* Heterogeneity exists in various types of datacenter infrastructure [13, 14]. Compute or storage hardware upgrades can potentially alter the behavior of the existing system [57, 81]. Network updates affect many factors, such as available link capacities, packet loss rates, and delay [82, 83].

- *Application diversity and workload dynamics.* Prior work has shown that there can be non-trivial differences among cloud applications [15, 59, 84]. In addition, cloud applications have increasingly adopted tighter and more frequent software update cycles [6, 85], including changes in architectures, bug fixes or patches, and even payloads.

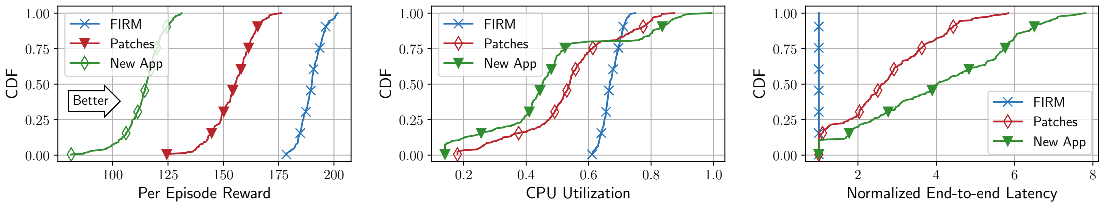
For instance: (a) In SLO-driven resource management, application performance and utilization differ significantly among heterogeneous workloads [1]. (b) In power management, diverse power consumption and workload sensitivity to core/uncore frequency require separate training of RL policies [16]. (c) In video streaming and network congestion control, different sets of traces have diverse payload characteristics and network environments [70] (e.g., dynamic link bandwidth, delay, and loss rate). Even with transfer learning (TL) [1], nontrivial retraining is needed to adapt to new workloads and environment shifts in each problem domain, which is a critical problem in making RL practical in production. Further, TL requires fine-grained environment clustering to identify the most appropriate model to transfer from, and requires saving one model per cluster.

Second, for the same application and environment, there could be slight changes (e.g.,



(a) OpenWhisk dataset. **(b)** CloudBandit dataset. **(c)** Estimation error.

Figure 2.7: Testing trained Sizeless model on datasets collected from unseen cloud platforms or new applications.



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

Figure 2.8: Testing trained FIRM model on unseen applications or application updates.

patches and rolling updates), unusual workload patterns not seen before (e.g., due to migration rollout), or traffic jitters. Without timely retraining, the online policy-serving performance of the RL agent fluctuates and leads to undesired degradation (e.g., SLO violations, low resource utilization, and network congestion). It is crucial to ensure robust online performance in case of environment or model uncertainty [86, 87].

Third, RL training is through trial and error [1, 65, 69], so worse-than-baseline or suboptimal decisions can be generated, especially at an early stage of training. Direct training in the production system leads to suboptimal performance and undesired SLO violations, while training in a simulator and then transitioning to the production system face the problem of poor generalization [88].

Characterization of Model Adaptation Requirements. 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 require a large amount of training data [57, 67]. To characterize such 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 in the two case studies are deferred to Appendices A.2 and A.3.

In RCS, we quantify the model performance degradation and show in Fig. 2.7(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 [64]). 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. Fig. 2.7(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.*

In this dissertation, we aim to bridge the gap and build a framework that can bring ML advances to production systems so that:

(1) The learned policy can handle complexity and dynamics with online telemetry data and reinforcement learning (addressed with a multi-level ML/RL framework in Chapter 3);

(2) The ML agents (which learn and operate the policies, controlling each system task) can co-exist on the same shared, multi-tenant cloud platform and reach convergence without interference (addressed with a virtual agent abstraction based on mean-field theory in Chapter 4); and

(3) The policy (ML model) can be trained or retrained safely and robustly while adapting to new applications and cloud environments seamlessly without significant retraining (addressed by leveraging meta-learning in Chapter 5);

CHAPTER 3: LEARNING CLOUD RESOURCE MANAGEMENT POLICIES

3.1 INTRODUCTION

User-facing latency-sensitive web services, like those at Netflix [24], Google [22], and Amazon [21], are increasingly built as microservices that execute on shared/multi-tenant compute resources either as virtual machines (VMs) or as containers (with containers gaining significant popularity of late). These microservices must handle diverse load characteristics while efficiently multiplexing shared resources in order to maintain service-level objectives (SLOs) like end-to-end latency. SLO violations occur when one or more "critical" microservice instances (defined in Section 2.2) experience load spikes (due to diurnal or unpredictable workload patterns) or shared-resource contention, both of which lead to longer than expected times to process requests, i.e., latency spikes [5, 6, 7, 89, 90, 91, 92, 93, 94, 95]. Thus, it is critical to efficiently multiplex shared resources among microservices to reduce SLO violations.

Traditional approaches (e.g., overprovisioning [96, 97], recurrent provisioning [98, 99], and autoscaling [47, 48, 49, 50, 51, 100, 101]) reduce SLO violations by allocating more CPUs and memory to microservice instances by using performance models, handcrafted heuristics (i.e., static policies), or machine-learning algorithms.

Unfortunately, these approaches suffer from two main problems. First, they fail to efficiently multiplex resources, such as caches, memory, I/O channels, and network links, at fine granularity, and thus may not reduce SLO violations. For example, in Fig. 3.1, the Kubernetes container-orchestration system [32] is unable to reduce the tail latency spikes arising from contention for a shared resource like memory bandwidth, as its autoscaling algorithms were built using heuristics that only monitor CPU utilization, which does not change much during the latency spike. Second, significant human effort and training are needed to build high-fidelity performance models (and related scheduling heuristics) of large-scale microservice deployments (e.g., queuing systems [50, 102]) that can capture low-level resource contention. Further, frequent microservice updates and migrations can lead to recurring human-expert-driven engineering efforts for model reconstruction.

FIRM Framework. This chapter addresses the above problems by presenting *FIRM*, a multilevel machine learning (ML) based resource management (RM) framework to manage shared resources among microservices at finer granularity to reduce resource contention and thus increase performance isolation and resource utilization. As shown in Fig. 3.1, FIRM performs better than a default Kubernetes autoscaler because FIRM adaptively scales up the

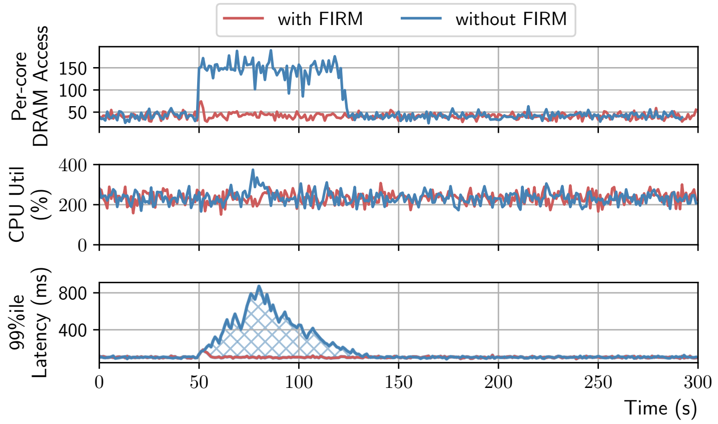


Figure 3.1: Latency spikes on microservices due to low-level resource contention.

microservice (by adding local cores) to increase the aggregate memory bandwidth allocation, thereby effectively maintaining the per-core allocation. FIRM leverages online telemetry data (such as request-tracing data and hardware counters) to capture the system state, and ML models for resource contention estimation and mitigation. Online telemetry data and ML models enable FIRM to adapt to workload changes and alleviate the need for brittle, handcrafted heuristics. In particular, FIRM uses the following ML models:

- *Support vector machine (SVM) driven detection and localization of SLO violations to individual microservice instances.* FIRM first identifies the "critical paths," and then uses per-critical-path and per-microservice-instance performance variability metrics (e.g., sojourn time [103]) to output a binary decision on whether or not a microservice instance is responsible for SLO violations.

- *Reinforcement learning (RL) driven mitigation of SLO violations that reduces contention on shared resources.* FIRM then uses resource utilization, workload characteristics, and performance metrics to make dynamic reprovisioning decisions, which include (1) increasing or reducing the partition portion or limit for a resource type, (2) scaling up/down, i.e., adding or reducing the amount of resources attached to a container, and (3) scaling out/in, i.e., scaling the number of replicas for services. By continuing to learn mitigation policies through reinforcement, FIRM can optimize for dynamic workload-specific characteristics.

Online Training for FIRM. We developed a *performance anomaly injection framework* that can artificially create resource scarcity situations in order to both train and assess the proposed framework. The injector is capable of injecting resource contention problems at a fine granularity (such as last-level cache and network devices) to trigger SLO violations. To enable rapid (re)training of the proposed system as the underlying systems [104] and workloads [90, 105, 106, 107] change in datacenter environments, FIRM uses *transfer learning*. That is, FIRM leverages transfer learning to train microservice-specific RL agents based on previous RL experience.

Contributions. To the best of our knowledge, this is the first work to provide an SLO violation mitigation framework for microservices by using fine-grained resource management in an application-architecture-agnostic way with multilevel ML models. Our main contributions are:

- *SVM-based SLO Violation Localization:* We present (in Section 3.2.2 and Section 3.2.3) an efficient way of localizing the microservice instances responsible for SLO violations by extracting critical paths and detecting anomaly instances in near-real time using telemetry data.

- *RL-based SLO Violation Mitigation:* We present (in Section 3.2.4) an RL-based resource contention mitigation mechanism that (a) addresses the large state space problem and (b) is capable of tuning tailored RL agents for individual microservice instances by using transfer learning.

- *Online Training & Performance Anomaly Injection:* We propose (in Section 3.2.6) a comprehensive performance anomaly injection framework to artificially create resource contention situations, thereby generating the ground-truth data required for training the aforementioned ML models.

- *Implementation & Evaluation:* We provide an open-source implementation of FIRM for the Kubernetes container-orchestration system [32]. We demonstrate and validate this implementation on four real-world microservice benchmarks [39, 42] (in Section 3.3).

Results. FIRM significantly outperforms state-of-the-art RM frameworks like Kubernetes autoscaling [32, 108] and additive increase multiplicative decrease (AIMD) based methods [52, 53].

- It reduces overall SLO violations by up to 16× compared with Kubernetes autoscaling, and 9× compared with the AIMD-based method, while reducing the overall requested CPU by as much as 62%.

- It outperforms the AIMD-based method by up to 9× and Kubernetes autoscaling by up to 30× in terms of the time to mitigate SLO violations.

- It improves overall performance predictability by reducing the average tail latencies up to 11×.
- It successfully localizes SLO violation root-cause microservice instances with 93% accuracy on average.

FIRM mitigates SLO violations without overprovisioning because of two main features. First, it models the dependency between low-level resources and application performance in an RL-based feedback loop to deal with uncertainty and noisy measurements. Second, it takes a two-level approach in which the online critical path analysis and the SVM model filter only those microservices that need to be considered to mitigate SLO violations, thus making the framework application-architecture-agnostic as well as enabling the RL agent to be trained faster.

3.2 FIRM FRAMEWORK DESIGN AND IMPLEMENTATION

In this section, we describe the overall architecture of the FIRM framework and its implementation.

(1) Based on the insight that resource contention manifests as dynamically evolving CPs, FIRM first detects CP changes and extracts critical microservice instances from them. It does so using the *Tracing Coordinator*, which is marked as ❶ in Fig. 1.2.[5] The tracing coordinator collects tracing and telemetry data from every microservice instance and stores them in a centralized graph database for processing. It is described in Section 3.2.1.

(2) The *Extractor* detects SLO violations and queries the Tracing Coordinator with collected real-time data (i) to extract CPs (marked as ❷ and described in Section 3.2.2) and (ii) to localize critical microservice instances that are likely causes of SLO violations (marked as ❸ and described in Section 3.2.3).

(3) Using the telemetry data collected in ❶ and the critical instances identified in ❸, FIRM makes mitigation decisions to scale and reprovision resources for the critical instances (marked as ❹). The policy used to make such decisions is automatically generated using RL. The RL agent jointly analyzes contextual information about resource utilization (i.e., low-level performance counter data collected from the CPU, LLC, memory, I/O, and network), performance metrics (i.e., per-microservice and

[5] Unless otherwise specified, ⊛ refers to annotations in Fig. 1.2.

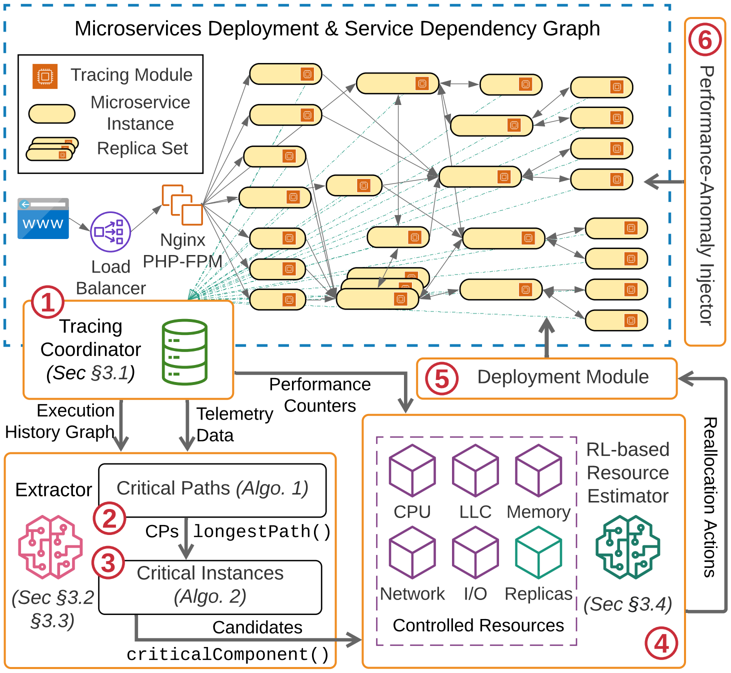


Figure 3.2: FIRM architecture overview.

end-to-end latency distributions), and workload characteristics (i.e., request arrival rate and composition) and makes mitigation decisions. The RL model and setup are described in Section 3.2.4.

(4) Finally, actions are validated and executed on the underlying Kubernetes cluster through the deployment module (marked as ❺ and described in Section 3.2.5).

(5) In order to train the ML models in the Extractor as well as the RL agent (i.e., to span the exploration-exploitation trade-off space), FIRM includes a performance anomaly injection framework that triggers SLO violations by generating resource contention with configurable intensity and timing. The anomaly injector is marked as ❻ and described in Section 3.2.6.

3.2.1 Tracing Coordinator

Distributed tracing is a method used to profile and monitor microservice-based applications to pinpoint causes of poor performance [109, 110, 111, 112, 113]. A *trace* captures the work done by each service along request execution paths, i.e., it follows the execution "route" of a request across microservice instances and records time, local profiling information, and RPC calls (e.g., source and destination services). The execution paths are combined to

Table 3.1: Collected telemetry data and sources.

cAdvisor [118] & **Prometheus** [119]
cpu_usage_seconds_total, memory_usage_bytes, fs_write/read_seconds,
fs_usage_bytes, network_transmit/receive_bytes_total, processes
Linux perf Subsystem [120]
offcore_response.*.{llc_hit.local_DRAM, llc_miss.local_DRAM}
offcore_response.*.{llc_hit.remote_DRAM, llc_miss.remote_DRAM}

form the *execution history graph* (see Section 2.2). The time spent by a single request in a microservice instance is called its *span*. The span is calculated based on the time when a request arrives at a microservice and when its response is sent back to the caller. Each span is the most basic single unit of work done by a microservice.

The FIRM tracing module's design is heavily inspired by Dapper [114] and its open-source implementations, e.g., Jaeger [110] and Zipkin [113]. Each microservice instance is coupled with an OpenTracing-compliant [115] *tracing agent* that measures spans. As a result, any new OpenTracing-compliant microservice can be integrated naturally into the FIRM tracing architecture. The Tracing Coordinator, i.e., ❶, is a stateless, replicable data-processing component that collects the spans of different requests from each tracing agent, combines them, and stores them in a graph database [116] as the execution history graph. The graph database allows us to easily store complex caller-callee relationships among microservices depending on request types, as well as to efficiently query the graph for critical path/component extraction (see Section 3.2.2 and Section 3.2.3). Distributed clock drift and time shifting are handled using the Jaeger framework. In addition, the Tracing Coordinator collects telemetry data from the systems running the microservices. The data collected in our experiments is listed in Table 3.1. The distributed tracing and telemetry collection overhead is indiscernible, i.e., we observed a <0.4% loss in throughput and a <0.15% loss in latency. FIRM had a maximum CPU overhead of 4.6% for all loads running in our experiments on the four benchmarks [39, 42]. With FIRM, the network in/out traffic without sampling traces increased by 3.4%/10.9% (in bytes); the increase could be less in production environments with larger message sizes [117].

3.2.2 Critical Path Extractor

The first goal of the FIRM framework is to quickly and accurately identify the CP based on the tracing and telemetry data described in the previous section. Recall from Def. 2.3 in Section 2.2 that a CP is the longest path in the request's execution history graph. Hence, changes in the end-to-end latency of an application are often determined by the slowest execution of one or more microservices on its CP.

We identify the CP in an execution history graph by using Alg. 3.1, which is a weighted longest path algorithm proposed to retrieve CPs in the microservices context. The algorithm needs to take into account the major communication and computation patterns in microservice architectures: (i) *parallel*, (ii) *sequential*, and (iii) *background* workflows.

- *Parallel workflows* are the most common way of processing requests in microservices. They are characterized by child spans of the same parent span that overlap with each other in the execution history graph, e.g., U, V, and T in Fig. 2.1(b). Formally, for two child spans i with start time st_i and end time et_i, and j with st_j, et_j of the same parent span p, they are called *parallel* if $(st_j < st_i < et_j) \ \vee \ (st_i < st_j < et_i)$.

- *Sequential workflows* are characterized by one or more child spans of a parent span that are processed in a serialized manner, e.g., U and I in Fig. 2.1(b). For two of p's child-spans i and j to be in a sequential workflow, the time $t_{i \to p} \leq t_{p \to j}$, i.e., i completes and sends its result to p before j does. Such sequential relationships are usually indicative of a *happens-before* relationship. However, it is impossible to ascertain the relationships merely by observing traces from the system. If, across a sufficient number of request executions, there is a violation of that inequality, then the services are not sequential.

- *Background workflows* are those that do not return values to their parent spans, e.g., W in Fig. 2.1(b). Background workflows are not part of CPs since no other span depends on their execution, but they may be considered responsible for SLO violations when FIRM's Extractor is localizing critical components (see Section 3.2.3). That is because background workflows may also contribute to the contention of underlying shared resources.

3.2.3 Critical Component Extractor

In each extracted CP, FIRM then uses an adaptive, data-driven approach to determine critical components (i.e., microservice instances). The overall procedure is shown in Alg. 3.2. The extraction algorithm first calculates per-CP and per-instance "features," which represent the performance variability and level of request congestion. Variability represents the single largest opportunity to reduce tail latency. The two features are then fed into an incremental SVM classifier to get binary decisions, i.e., on whether that instance should have its resources re-provisioned or not. The approach is a dynamic selection policy that is in contrast to

Algorithm 3.1 Critical Path Extraction

```
Require: Microservice execution history graph $G$
    Attributes: $childNodes$, $lastReturnedChild$
procedure LongestPath($G$, $currentNode$)
    $path \leftarrow \varnothing$
    $path$.add($currentNode$)
    if $currentNode.childNodes$ == None then
        Return $path$
    end if
    $lrc \leftarrow currentNode.lastReturnedChild$
    $path$.extend(LongestPath($G$, $lrc$))
    for each $cn$ in $currentNode.childNodes$ do
        if $cn$.happensBefore($lrc$) then
            $path$.extend(LongestPath($G$, $cn$))
        end if
    end for
    Return $path$
end procedure
```

Algorithm 3.2 Critical Component Extraction

```
Require: Critical Path $CP$, Request Latencies $T$
procedure CriticalComponent($G$, $T$)
    $candidates \leftarrow \varnothing$
    $T_{CP} \leftarrow T.getTotalLatency()$                    ▷ Vector of CP latencies
    for $i \in CP$ do
        $T_i \leftarrow T.getLatency(i)$
        $T_{99} \leftarrow T_i.percentile(99)$
        $T_{50} \leftarrow T_i.percentile(50)$
        $RI \leftarrow PCC(T_i, T_{CP})$                       ▷ Relative Importance
        $CI \leftarrow T_{99}/T_{50}$                          ▷ Congestion Intensity
        if $SVM.classify(RI, CI) == True$ then
            $candidates.append(i)$
        end if
    end for
    Return $candidates$
end procedure
```

static policies, as it can classify critical and noncritical components adapting to dynamically changing workloads and variation patterns.

In order to extract those microservice instances that are potential candidates for SLO violations, we argue that it is critical to know both the variability of the end-to-end latency (i.e., per-CP variability) and the variability caused by congestion in the service queues of each individual microservice instances (i.e., per-instance variability).

Per-CP Variability: Relative Importance. Relative importance [121, 122, 123] is a metric that quantifies the strength of the relationship between two variables. For each critical path CP, its end-to-end latency is given by $T_{CP} = \sum_{i \in CP} T_i$, where T_i is the latency of microservice i. Our goal is to determine the contribution that the variance of each variable T_i makes toward explaining the total variance of T_{CP}. To do so, we use the Pearson correlation coefficient [124] (also called zero-order correlation), i.e., $PCC(T_i, T_{CP})$, as the measurement, and hence the resulting statistic is known as the variance explained [125]. The sum of $PCC(T_i, T_{CP})$ over all microservice instances along the CP is 1, and the relative importance values of microservices can be ordered by $PCC(T_i, T_{CP})$. The larger the value is, the more variability it contributes to the end-to-end CP variability.

Per-Instance Variability: Congestion Intensity. For each microservice instance in a CP, congestion intensity is defined as the ratio of the 99th percentile latency to the median latency. Here, we chose the 99th percentile instead of the 70th or 80th percentile to target the tail latency behavior. The chosen ratio explains per-instance variability by capturing the congestion level of the request queue so that it can be used to determine whether it is necessary to scale. For example, a higher ratio means that the microservice could handle only a subset of the requests, but the requests at the tail are suffering from congestion issues in the queue. On the other hand, microservices with lower ratios handle most requests normally, so scaling does not help with performance gain. Consequently, microservice instances with higher ratios have a greater opportunity to achieve performance gains in terms of tail latency by taking scale-out or reprovisioning actions.

Implementation. The logic of critical path extraction is incorporated into the construction of spans, i.e., as the algorithm proceeds (Alg. 3.1), the order of tracing construction is also from the root node to child nodes recursively along paths in the execution history graph. Sequential, parallel, and background workflows are inferred from the parent-child relationships of spans. Then, for each CP, we calculate feature statistics and feed them into an incremental SVM classifier [126, 127] implemented using stochastic gradient descent optimization and RBF kernel approximation by `scikit-learn` libraries [128]. Triggered by detected SLO violations, both critical path extraction and critical component extraction are stateless and multithreaded; thus, the workload scales with the size of the microservice application and the cluster. They together constitute FIRM's extractor (i.e., ❷ and ❸). Experiments (Section 3.3.2) show that it reports SLO violation candidates with feasible accuracy and achieves completeness with Section 3.2.4 by choosing a threshold with a reasonable false-positive rate.

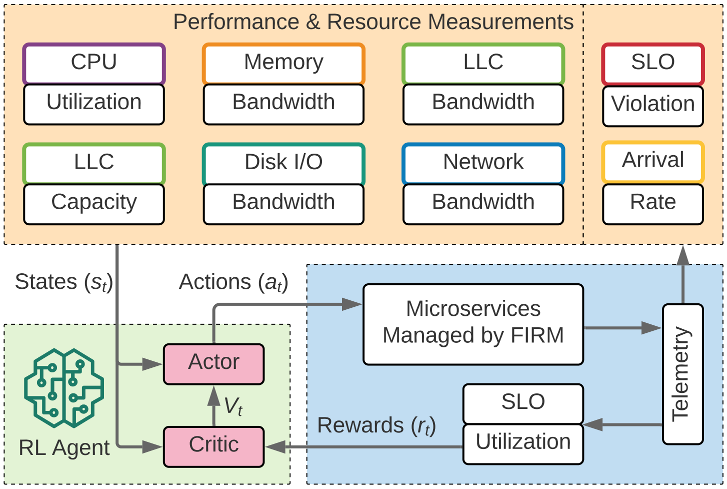


Figure 3.3: Model-free actor-critic RL framework for estimating resources in a microservice instance.

3.2.4 SLO Violation Mitigation Using RL

Given the list of critical service instances, FIRM's Resource Estimator, i.e., ❹, is designed to analyze resource contention and provide reprovisioning actions for the cluster manager to take. FIRM estimates and controls a fine-grained set of resources, including CPU time, memory bandwidth, LLC capacity, disk I/O bandwidth, and network bandwidth. It makes decisions on scaling each type of resource or the number of containers by using measurements of tracing and telemetry data (see Table 3.1) collected from the Tracing Coordinator. When jointly analyzed, such data provides information about (i) shared-resource interference, (ii) workload rate variation, and (iii) request type composition.

FIRM leverages reinforcement learning (RL) to optimize resource management policies for long-term reward in dynamic microservice environments. We next give a brief RL primer before presenting FIRM's RL model.

RL Primer. An RL agent solves a *sequential decision-making problem* (modeled as a Markov decision process) by interacting with an environment. At each discrete time step t, the agent observes a *state of the environment* $s_t \in S$, and performs an *action* $a_t \in A$ based on its *policy* $\pi_\theta(s)$ (parameterized by θ), which maps *state space* S to *action space* A. At the following time step $t+1$, the agent observes an *immediate reward* $r_t \in R$ given by a reward function $r(s_t, a_t)$; the immediate reward represents the loss/gain in transitioning from s_t to s_{t+1} because of action a_t. The tuple (s_t,a_t,r_t,s_{t+1}) is called one *transition*. The agent's goal is to optimize the policy π_θ so as to maximize the expected *cumulative discounted reward* (also called the value function) from the start distribution $J = \mathbb{E}[G_1]$, where the return from a state G_t is defined to be $\sum_{k=0}^{T} \gamma^k r_{t+k}$. The discount factor $\gamma \in (0, 1]$ penalizes the predicted future rewards.

Two main categories of approaches are proposed for policy learning: value-based methods and policy-based methods [129]. In value-based methods, the agent learns an estimate of the optimal value function and approaches the optimal policy by maximizing it. In policy-based methods, the agent directly tries to approximate the optimal policy.

Why RL? Existing performance-modeling-based [49, 50, 51, 52, 53, 54, 55] or heuristic-based approaches [47, 48, 130, 131, 132] suffer from model reconstruction and retraining problems because they do not address dynamic system status. Moreover, they require expert knowledge, and it takes significant effort to devise, implement, and validate their understanding of the microservice workloads as well as the underlying infrastructure. RL, on the other hand, is well-suited for learning resource reprovisioning policies, as it provides a tight feedback loop for exploring the action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules). It allows direct learning from the actual workload and operating conditions to understand how adjusting low-level resources affects application performance. In particular, FIRM utilizes the deep deterministic policy gradient (DDPG) algorithm [133], which is a *model-free*, *actor-critic* RL framework (shown in Fig. 3.3). Further, FIRM's RL formulation provides two distinct advantages:

(1) Model-free RL does not need the ergodic distribution of states or the environment dynamics (i.e., transitions between states), which are difficult to model precisely. When microservices are updated, the simulations of state transitions used in model-based RL are no longer valid.

(2) The Actor-critic framework combines policy-based and value-based methods (i.e., consisting of an actor-net and a critic-net as shown in Fig. 3.4), and that is suitable for continuous stochastic environments, converges faster, and has lower variance [134].

Learning the Optimal Policy. DDPG's policy learning is an actor-critic approach. Here, the "critic" estimates the *value function* (i.e., the expected value of cumulative discounted reward under a given policy), and the "actor" updates the policy in the direction suggested by the critic. The critic's estimation of the expected return allows the actor to update with gradients that have lower variance, thus speeding up the learning process (i.e., achieving convergence). We further assume that the actor and critic are represented as deep neural networks. DDPG also solves the issue of dependency between samples and makes use of hardware optimizations by introducing a *replay buffer*, which is a finite-sized cache $\mathcal{D}$ that stores transitions (s_t,a_t,r_t,s_{t+1}). Parameter updates are based on a mini-batch of size N sampled from the reply buffer. The pseudocode of the training algorithm is shown in

Algorithm 3.3 DDPG Training

```
Randomly init $Q_w(s,a)$ and $\pi_\theta(a|s)$ with weights $w$ & $\theta$.
Init target network $Q'$ and $\pi'$ with $w' \leftarrow w$ & $\theta' \leftarrow \theta$
Init replay buffer $\mathcal{D} \leftarrow \varnothing$
for episode = 1, $M$ do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for $t = 1, T$ do
        Select and execute action $a_t = \pi_\theta(s_t) + \mathcal{N}_t$
        Observe reward $r_t$ and new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
        Sample $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $\mathcal{D}$
        Update critic by minimizing the loss $\mathcal{L}(w)$
        Update actor by sampled policy gradient $\nabla_\theta J$
        $w' \leftarrow \gamma w + (1-\gamma) w'$
        $\theta' \leftarrow \gamma\theta + (1-\gamma)\theta'$
    end for
end for
```

Algorithm 3.3. RL training proceeds in episodes and each episode consists of T time steps. At each time step, both actor and critic neural nets are updated once.

In the critic, the value function $Q_w(s_t, a_t)$ with parameter w and its corresponding loss function are defined as:

$$Q_w(s_t, a_t) = \mathbb{E}[r(s_t, a_t) + \gamma Q_w(s_{t+1}, \pi(s_{t+1}))] \tag{3.1}$$

$$\mathcal{L}(w) = \frac{1}{N}\sum_i (r_i + \gamma Q'_{w'}(s_{i+1}, \pi'_{\theta'}(s_{i+1})) - Q_w(s_i, a_i))^2 \tag{3.2}$$

The target networks $Q'_{w'}(s,a)$ and $\pi'_{\theta'}(s)$ are introduced in DDPG to mitigate the problem of instability and divergence when one is directly implementing deep RL agents. In the actor component, DDPG maintains a parametrized actor function $\pi_\theta(s)$, which specifies the current policy by deterministically mapping states to a specific action. The actor is updated as follows:

$$\nabla_\theta J = \frac{1}{N}\sum_i \nabla_a Q_w(s = s_i, a = \pi(s_i)) \nabla_\theta \pi_\theta(s = s_i) \tag{3.3}$$

Problem Formulation. To estimate resources for a microservice instance, we formulate a sequential decision-making problem that can be solved by the above RL framework. Each microservice instance is deployed in a separate container with a tuple of resource limits

$RLT = (RLT_{cpu}, RLT_{mem}, RLT_{llc}, RLT_{io}, RLT_{net})$, since we are considering CPU utilization, memory bandwidth, LLC capacity, disk I/O bandwidth, and network bandwidth as our resource model.[6] This limit for each type of resource is predetermined (usually overprovisioned) before the microservices are deployed in the cluster and later controlled by FIRM.

At each time step t, utilization RU_t for each type of resource is retrieved using performance counters as telemetry data in ❶. In addition, FIRM's Extractor also collects current latency, request arrival rate, and request type composition (i.e., percentages of each type of request). Based on these measurements, the RL agent calculates the states listed in Table 3.2 and described below.

- *SLO maintenance ratio* (SM_t) is defined as `SLO_latency/ current_latency` if the microservice instance is determined to be the culprit. If no message arrives, it is assumed that there is no SLO violation ($SM_t = 1$).
- *Workload changes* (WC_t) is defined as the ratio of the arrival rates at the current and previous time steps.
- *Request composition* (RC_t) is defined as a unique value encoded from an array of request percentages by using `numpy.ravel_multi_index()` [135].

For each type of resource i, there is a predefined resource upper limit $\hat{R}_i$ and a lower limit $\underaccent{\check}{R}_i$ (e.g., the CPU time limit cannot be set to 0). The actions available to the RL-agent are to set $RLT_i \in [\hat{R}_i, \underaccent{\check}{R}_i]$. If the amount of resources reaches the total available amount, then a scale-out operation is needed. Similarly, if the resource limit is below the lower bound, a scale-in operation is needed. The CPU resources serve as one exception to the above procedure: it would not improve the performance if the CPU utilization limit were higher than the number of threads created for the service.

The goal of the RL agent is, given a time duration t, to determine an optimal policy π_t that results in as few SLO violations as possible (i.e., $\min_{\pi_t} SM_t$) while keeping the resource utilization/limit as high as possible (i.e., $\max_{\pi_t} RU_t/RLT_t$). Based on both objectives, the reward function is then defined as $r_t = \alpha \cdot SM_t \cdot |\mathcal{R}| + (1-\alpha) \cdot \sum_i^{|\mathcal{R}|} RU_i/RLT_i$, where $\mathcal{R}$ is the set of resources.

Transfer Learning. Using a tailored RL agent for every microservice instead of using the shared RL agent should improve resource reprovisioning efficiency, as the model would be more sensitive to application characteristics and features. However, such an approach is hard to justify in practice (i.e., for deployment) because of the time required to train such

[6] The resource limit for the CPU utilization of a container is the smaller value between $\hat{R}_i$ and the number of threads × 100 (i.e., based on the multi-threading implementation).

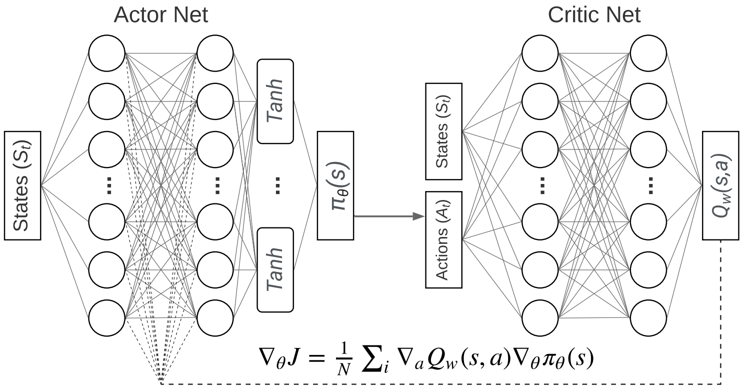


Figure 3.4: Architecture of actor-critic nets.

tailored models for user workloads, which might have significant churn. FIRM addresses the problem of rapid model training by using transfer learning in the domain of RL [136, 137, 138], whereby agents for SLO violation mitigation can be trained for either the general case (i.e., any microservices) or the specialized case (i.e., "transferred" to the behavior of individualized microservices). The pre-trained model used in the specialized case is called the base model or the source model. That approach is possible because prior understanding of a problem structure helps one solve similar problems quickly, with the remaining task being to understand the behavior of updated microservice instances. Related work on base model selection and task similarity can be found in [136, 137], but the base model that FIRM uses for transfer learning is always the RL model learned in the general case because it has been shown in evaluation to be comparable with specialized models. We demonstrate the efficacy of transfer learning in our evaluation described in Section 3.3. The RL model that FIRM uses is designed to scale since both the state space and the action space are independent of the size of the application or the cluster. In addition to having the general case RL agent, the FIRM framework also allows for the deployment of specialized per-microservice RL agents.

Table 3.2: State-action space of the RL agent.

State Space (s_t)
SLO Maintenance Ratio (SM_t), Workload Changes (WC_t), Request Composition (RC_t), Resource Utilization (RU_t)
Action Space (a_t)
Resource Limits $RLT_i(t), i \in \{$CPU, Mem, LLC, IO, Net$\}$

Implementation Details. We implemented the DDPG training algorithm and the actor-critic networks using `PyTorch` [139]. The critic net contains two fully connected hidden layers with 40 hidden units, all using the ReLU activation function. The first two hidden layers of the actor net are fully connected and both use ReLU as the activation function while

Table 3.3: RL training parameters.

Parameter	**Value**
# Time Steps × # Minibatch	300 × 64
Size of Replay Buffer	10^5
Learning Rate	Actor (3×10^{-4}), Critic (3×10^{-3})
Discount Factor	0.9
Soft Update Coefficient	2×10^{-3}
Random Noise	μ (0), σ (0.2)
Exploration Factor	ϵ (1.0), ϵ-decay (10^{-6})

the last layer uses Tanh as the activation function. The actor network has 8 inputs and 5 outputs, while the critic network has 23 inputs and 1 output. The actor and critic networks are shown in Fig. 3.4, and their inputs and outputs are listed in Table 3.2. We chose that setup because adding more layers and hidden units does not increase performance in our experiments with selected microservice benchmarks; instead, it slows down training speed significantly. Hyperparameters of the RL model are listed in Table 3.3. We set the time step for training the model to be 1 second, which is sufficient for action execution (see Table 3.5). The latencies of each RL training update and inference step are 73 ± 10.9 ms and 1.2 ± 0.4 ms, respectively. The average CPU and memory usages of the Kubernetes pod during the training stage are 210 millicores and 192 Mi, respectively.

3.2.5 Action Execution

FIRM's Deployment Module, i.e., ❺, verifies the actions generated by the RL agent and executes them accordingly. Each action on scaling a specific type of resource is limited by the total amount of the resource available on that physical machine. FIRM assumes that machine resources are unlimited and thus does not have admission control or throttling. If an action leads to oversubscribing of a resource, then it is replaced by a scale-out operation.

- *CPU Actions:* Actions on scaling CPU utilization are executed through modification of `cpu.cfs_period_us` and `cpu.cfs_quota_us` in the `cgroups` CPU subsystem.
- *Memory Actions:* We use Intel MBA [140] and Intel CAT [141] technologies to control the memory bandwidth and LLC capacity of containers, respectively.[7]
- *I/O Actions:* For I/O bandwidth, we use the `blkio` subsystem in `cgroups` to control input/output access to disks.

[7] Our evaluation on IBM Power systems (see Section 3.3) did not use these actions because of a lack of hardware support. OS support or software partitioning mechanisms [142, 143] can be applied; we leave that to future work.

Table 3.4: Types of performance anomalies injected causing SLO violations.

Performance Anomaly Types	**Tools/Benchmarks**
Workload Variation	`wrk2` [145]
Network Delay	`tc` [146]
CPU Utilization	iBench [147], `stress-ng` [148]
LLC Bandwidth & Capacity	iBench, `pmbw` [149]
Memory Bandwidth	iBench [147], `pmbw` [149]
I/O Bandwidth	Sysbench [150]
Network Bandwidth	`tc` [146], Trickle [151]

- *Network Actions:* For network bandwidth, we use the Hierarchical Token Bucket (HTB) [144] queueing discipline in Linux Traffic Control. Egress `qdiscs` can be directly shaped by using HTB. Ingress `qdiscs` are redirected to the virtual device `ifb` interface and then shaped through the application of egress rules.

3.2.6 Performance Anomaly Injector

We accelerate the training of the machine learning models in FIRM's Extractor and the RL agent through performance anomaly injections. The injection provides the ground truth data for the SVM model, as the injection targets are controlled and known from the campaign files. It also allows the RL agent to quickly span the space of adverse resource contention behavior (i.e., the exploration-exploitation trade-off in RL). That is important, as real-world workloads might not experience all adverse situations within a short training time. We implemented a performance anomaly injector, i.e., ❻, in which the injection targets, type of anomaly, injection time, duration, patterns, and intensity are configurable. The injector is designed to be bundled into the microservice containers as a file-system layer; the binaries incorporated into the container can then be triggered remotely during the training process. The injection campaigns (i.e., how the injector is configured and used) for the injector will be discussed in Section 3.3. The injector comprises seven types of performance anomalies that can cause SLO violations. They are listed in Table 3.4 and described below.

Workload Variation. We use an HTTP benchmarking tool `wrk2` as the workload generator. It performs multithreaded, multiconnection HTTP request generation to simulate client-microservice interaction. The request arrival rate and distribution can be adjusted to break the predefined SLOs.

Network Delay. We use Linux traffic control (`tc`) to add simulated delay to network packets. Given the mean and standard deviation of the network delay latency, each network packet is delayed following a normal distribution.

CPU Utilization. We implement the CPU stressor based on iBench and `stree-ng` to exhaust a specified level of CPU utilization on a set of cores by exercising floating point, integer, bit manipulation, and control flows.

LLC Bandwidth & Capacity. We use iBench and `pmbw` to inject interference on the Last Level Cache (LLC). For bandwidth, the injector performs streaming accesses in which the size of the accessed data is tuned to the parameters of the LLC. For capacity, it adjusts intensity based on the size and associativity of the LLC to issue random accesses that cover the LLC capacity.

Memory Bandwidth. We use iBench and `pmbw` to generate memory bandwidth contention. It performs serial memory accesses (of configurable intensity) to a small fraction of the address space. Accesses occur in a relatively small fraction of memory in order to decouple the effects of contention in memory bandwidth from contention in memory capacity.

I/O Bandwidth. We use Sysbench to implement the file I/O workload generator. First, it creates test files that are larger than the size of the system RAM. Then, it adjusts the number of threads, read/write ratio, and sleeping/working ratio to meet a specified level of I/O bandwidth. We also use Tricle to limit the upload/download rate of a specific microservice instance.

Network Bandwidth. We use Linux traffic control (`tc`) to limit egress network bandwidth. For ingress network bandwidth, an intermediate function block (`ifb`) pseudo interface is set up, and inbound traffic is directed through that. In that way, the inbound traffic then becomes schedulable by the egress `qdisc` on the `ifb` interface, so the same rules for egress can be applied directly to ingress.

3.3 EVALUATION

This section discusses our experimental evaluation of the FIRM system and is organized as follows. In Section 3.3.1, we describe the experimental setup and the comparison baselines. Then, in Section 3.3.2, we evaluate the critical component localization capabilities of FIRM in analyzing SLO violations, and in Section 3.3.3, we evaluate the RL training and SLO violation mitigation performance. Finally, we demonstrate the end-to-end performance of FIRM on Kubernetes regarding the cluster utilization and microservices application performance.

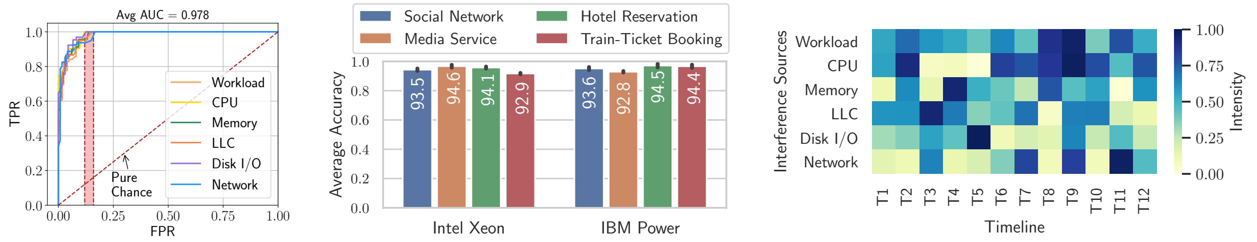
3.3.1 Experimental Setup

Benchmark Applications. We evaluated FIRM on a set of end-to-end interactive and responsive real-world microservice benchmarks: (i) DeathStarBench [39], consisting of *Social Network*, *Media Service*, and *Hotel Reservation* microservice applications, and (ii) Train-Ticket [152], consisting of the *Train-Ticket Booking Service*. *Social Network* implements a broadcast-style social network with unidirectional follow relationships whereby users can publish, read, and react to social media posts. *Media Service* provides functionalities such as reviewing, rating, renting, and streaming movies. *Hotel Reservation* is an online hotel reservation site for browsing hotel information and making reservations. *Train-Ticket Booking Service* provides typical train-ticket booking functionalities, such as ticket inquiry, reservation, payment, change, and user notification. These benchmarks contain 36, 38, 15, and 41 unique microservices, respectively; cover all workflow patterns (see Section 3.2.2); and use various programming languages including Java, Python, Node.js, Go, C/C++, Scala, PHP, and Ruby. All microservices are deployed in separate Docker containers.

System Setup. We validated our design by implementing a prototype of FIRM that used Kubernetes [32] as the underlying container orchestration framework. We deployed the four microservice benchmarks with FIRM separately on a Kubernetes cluster of 15 two-socket physical nodes without specifying any anti-colocation rules. Each server consists of 56–192 CPU cores and RAM that varies from 500 GB to 1000 GB. Nine of the servers use Intel x86 Xeon E5s and E7s processors, while the remaining ones use IBM ppc64 Power8 and Power9 processors. All machines run Ubuntu 18.04.3 LTS with Linux kernel version 4.15.

Load Generation. We drove the services with various open-loop asynchronous workload generators [145] to represent an active production environment [153, 154, 155]. We uniformly generated workloads for every request type across all microservice benchmarks. The parameters for the workload generators were the same as those for DeathStarBench (which we applied to Train-Ticket as well), and varied from predictable constant, diurnal, distributions such as Poisson, to unpredictable loads with spikes in user demand. The workload generators and the microservice benchmark applications were never co-located (i.e., they executed on different nodes in the cluster). To control the variability in our experiments, we disabled all other user workloads on the cluster.

Injection and Comparison Baselines. We used our performance anomaly injector (see Section 3.2.6) to inject various types of performance anomalies into containers uniformly at random with configurable injection timing and intensity. Following the common way to study resource interference, our experiments on SLO violation mitigation with anomalies were designed to be comprehensive by covering the worst-case scenarios, given the random and nondeterministic nature of shared-resource interference in production environments [7, 156]. Unless otherwise specified, (i) the anomaly injection time interval was in an exponential distribution with $\lambda = 0.33s^{-1}$, and (ii) the anomaly type and intensity were selected uniformly at random. We implemented two baseline approaches: (a) the Kubernetes autoscaling mech-



(a) ROC under single-anomaly injection. **(b)** Average accuracy under multi-anomaly injection. **(c)** Anomaly injection intensity and timing during one example run.

Figure 3.5: Critical Component Localization Performance: (a) ROC curves for detection accuracy; (b) Variation of localization accuracies across processor architectures; (c) Anomaly-injection intensity, types, and timing.

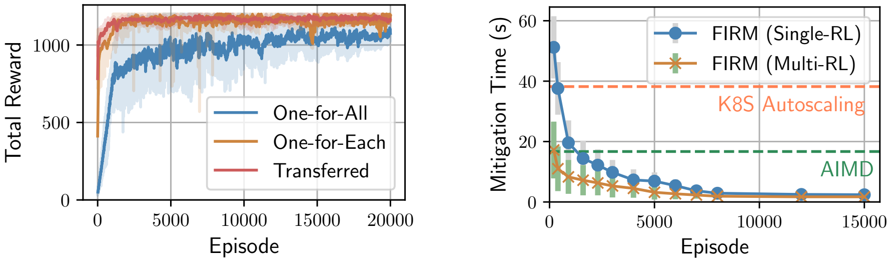
anism [108] and (b) an AIMD-based method [52, 53] to manage resources for each container. Both approaches are rule-based autoscaling techniques.

3.3.2 Critical Component Localization

Here, we use the techniques presented in Section 3.2.2 and Section 3.2.3 to study the effectiveness of FIRM in identifying the microservices that are most likely to cause application performance SLO violations.

Single anomaly localization. We first evaluated how well FIRM localizes the microservice instances that are responsible for SLO violations under different types of single-anomaly injections. For each type of performance anomaly and each type of request, we gradually increased the intensity of injected resource interference and recorded end-to-end latencies. The intensity parameter was chosen uniformly at random between [start-point, end-point], where the start-point is the intensity that starts to trigger SLO violations, and the end-point is the intensity when either the anomaly injector has consumed all possible resources or over 80% of user requests have been dropped or returned time. Fig. 3.5(a) shows the receiver operating characteristic (ROC) curve of root cause localization. The ROC curve captures the relationship between the false-positive rate (x-axis) and the true-positive rate (y-axis). The closer to the upper-left corner the curve is, the better the performance. We observe that the localization accuracy of FIRM, when subject to different types of anomalies, does not vary significantly. In particular, FIRM's Extractor module achieved near 100% true-positive rate, when the false-positive rate was between $[0.12, 0.16]$.

Multi-anomaly localization. There is no guarantee that only one resource contention will happen at a time under dynamic datacenter workloads [90, 105, 106, 107] and therefore, we also studied the container localization performance under multi-anomaly injections and



(a) Total reward. **(b)** SLO mitigation time.

Figure 3.6: Learning curve showing total reward during training and SLO mitigation performance.

compared machines with two different processor ISAs (x86 and ppc64). An example of the intensity distributions of all the anomaly types used in this experiment is shown in Fig. 3.5(c). The experiment was divided into time windows of 10 s, i.e., T_i from Fig. 3.5(c)). At each time window, we picked the injection intensity of each anomaly type uniformly at random with a range [0,1]. Our observations are reported in Fig. 3.5(b). The average accuracy for localizing critical components in each application ranged from 92% to 94%. The overall average localization accuracy was 93% across four microservice benchmarks. Overall, we observe that the accuracy of the Extractor did not differ between the two sets of processors.

3.3.3 RL Training & SLO Violation Mitigation

To understand the convergence behavior of FIRM's RL agent, we trained three RL models that were subjected to the same sequence of performance anomaly injections (described in Section 3.3.1). The three RL models are: (i) a common RL agent for all microservices (one-for-all), (ii) a tailored RL agent for a particular microservice (one-for-each), and (iii) a transfer-learning-based RL agent. RL training proceeds in episodes (iterations). We set the number of time steps in a training episode to be 300 (see Table 3.3), but for the initial stages, we terminate the RL exploration early so that the agent could reset and try again from the initial state. We did so because the initial policies of the RL agent are unable to mitigate SLO violations. Continuously injecting performance anomalies causes user requests to drop, and thus, only a few request traces were generated to feed the agent. As the training progressed, the agent improved its resource estimation policy and could mitigate SLO violations in less time. At that point (around 1000 episodes), we linearly increased the number of time steps to let the RL agent interact with the environment longer before terminating the RL exploration and entering the next iteration.

We trained the abovementioned three RL models on the Train-Ticket benchmark. We studied the generalization of the RL model by evaluating the end-to-end performance of FIRM on the DeathStarBench benchmarks. Thus, we used DeathStarBench as a validation set in our experiments. Fig. 3.6(a) shows that as the training proceeded, the agent was getting better at mitigation, and thus the moving average of episode rewards was increasing. The initial steep increase benefits from early termination of episodes and parameter exploration. Transfer-learning-based RL converged even faster (around 2000 iterations[8]) because of parameter sharing. The one-for-all RL required more iterations to converge (around 15000 iterations) and had a slightly lower total reward (6% lower compared with one-for-each RL) during training.

In addition, higher rewards, for which the learning algorithm explicitly optimizes, correlate with improvements in SLO violation mitigation (see Fig. 3.6(b)). For models trained in every 200 episodes, we saved the checkpoint of parameters in the RL model. Using the parameter, we evaluated the model snapshot by injecting performance anomalies (described in Section 3.3.1) continuously for one minute and observed when SLO violations were mitigated. Fig. 3.6(b) shows that FIRM with either a single-RL agent (one-for-all) or a multi-RL agent (one-for-each) improved with each episode in terms of the SLO violation mitigation time. The starting policy at iteration 0–900 was no better than the Kubernetes autoscaling approach, but after around 2500 iterations, both agents were better than either Kubernetes autoscaling or the AIMD-based method. Upon convergence, FIRM with a single-RL agent achieved a mitigation time of 1.7 s on average, which outperformed the AIMD-based method by up to 9× and Kubernetes autoscaling by up to 30× in terms of the time to mitigate SLO violations.

3.3.4 End-to-End Performance

Here, we show the end-to-end performance of FIRM and its generalization by further evaluating it on DeathStarBench benchmarks based on the hyperparameter tuned during training with the Train-Ticket benchmark. To understand the 10–30× improvement demonstrated above, we measured the 99th percentile end-to-end latency when the microservices were being managed by the two baseline approaches and by FIRM. Fig. 3.7(a) shows the cumulative distribution of the end-to-end latency. We observed that the AIMD-based method, albeit simple, outperforms the Kubernetes autoscaling approach by 1.7× on average and by 1.6× in the worst case. In contrast, FIRM:

(1) Outperformed both baselines by up to 6× and 11×, which leads to 9× and 16× fewer

[8] 1000 iterations correspond to roughly 30 minutes with each iteration consisting of 300 timesteps.

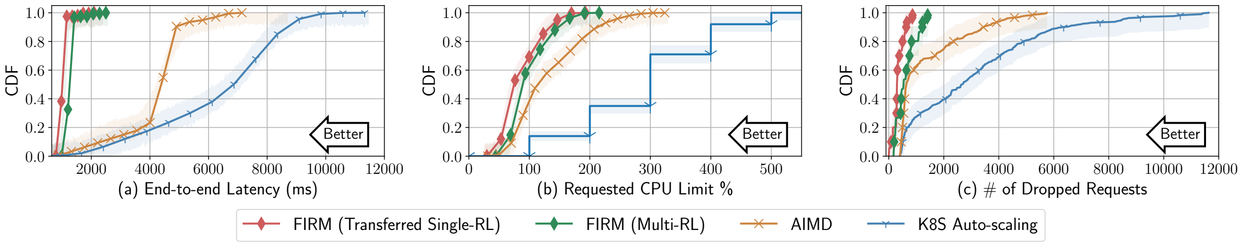


Figure 3.7: Performance comparisons (CDFs) of end-to-end latency, requested CPU limit, and the number of dropped requests.

SLO violations;

(2) Lowered the overall requested CPU limit by 29–62%, as shown in Fig. 3.7(b), and increased the average cluster-level CPU utilization by up to 33%; and

(3) Reduced the number of dropped or timed out user requests by up to 8× as shown in Fig. 3.7(c).

FIRM can provide these benefits because it detects SLO violations accurately and addresses resource contention before SLO violations can propagate. By interacting with dynamic microservice environments under complicated loads and resource allocation scenarios, FIRM's RL agent dynamically learns the policy, and hence outperforms heuristics-based approaches.

3.4 DISCUSSION

Necessity and Challenges of Modeling Low-level Resources. Recall from Section 2.2 that modeling of resources at a fine granularity is necessary, as it allows for better performance without overprovisioning. It is difficult to model the dependence between low-level resource requirements and quantifiable performance gain while dealing with uncertain and noisy measurements [157, 158]. FIRM addresses the issue by modeling that dependency in an RL-based feedback loop, which automatically explores the action space to generate optimal policies without human intervention.

Why a Multilevel ML Framework? A model of the states of all microservices that is fed as the input to a single large ML model [100, 159] leads to (i) state-action space explosion issues that grow with the number of microservices, thus increasing the training time; and (ii) dependence between the microservice architecture and the ML model, which sacrifices the generality. FIRM addresses those problems by incorporating a two-level ML framework. The first-level ML model uses SVM to filter the microservice instances responsible for SLO violations, thereby reducing the number of microservices that need to be considered in

Table 3.5: Average latency for resource management operations.

Operation	Partition (Scale Up/Down)					Container Start	
	CPU	Mem	LLC	I/O	Net	Warm	Cold
Mean (ms)	2.1	42.4	39.8	2.3	12.3	45.7	2050.8
Std Dev (ms)	0.3	11.0	9.2	0.4	1.1	6.9	291.4

mitigating SLO violations. That enables the second-level ML model, the RL agent, to be trained faster and removes dependence on the application architecture. That, in turn, helps avoid RL model reconstruction/retraining.

Lower Bounds on Manageable SLO Violation Duration for FIRM. As shown in Table 3.5, the operations to scale resources for microservice instances take 2.1–45.7 ms. Thus, that is the minimum duration of latency spikes that any RM approach can handle. For transient SLO violations, which last shorter than the minimum duration, the action generated by FIRM will always miss the mitigation deadline and can potentially harm overall system performance. Worse, it may lead to oscillations between scaling operations. Predicting the spikes before they happen, and proactively taking mitigation actions can be a solution. However, it is a generally-acknowledged difficult problem, as microservices are dynamically evolving, in terms of both load and architectural design, which is subject to our future work.

Limitations. FIRM has several limitations that can be addressed in future work. First, FIRM currently focuses on resource interference caused by real workload demands. However, FIRM lacks the ability to detect application bugs or misconfigurations, which may lead to failures such as memory leaks. Allocating more resources to such microservice instances may harm the overall resource efficiency. Other sources of SLO violations, including global resource sharing (e.g., network switches or global file systems) and hardware causes (e.g., power-saving energy management), are also beyond FIRM's scope. Second, the scalability of FIRM is limited by the maximum scalability of the centralized graph database, and the boundary caused by the network traffic telemetry overhead. (Recall the lower bound on the SLO violation duration.) Third, implementing FIRM's tracing module based on sidecar proxies (i.e., service meshes) [160] can help minimize application instrumentation and has wider support of programming languages. Lastly, FIRM is limited by the number of partitions that Intel CAT/MBA provides, and we address this limitation in [161].

3.5 RELATED WORK

SLO violations in cloud applications and microservices are a popular and well-researched topic. We categorize prior work into two buckets: root cause analyzers and autoscalers. Both rely heavily on the collection of tracing and telemetry data.

Tracing and Probing for Microservices. Tracing for large-scale microservices (essentially distributed systems) helps understand the path of a request as it propagates through the components of a distributed system. Tracing requires either application-level instrumentation [109, 110, 111, 112, 113, 114, 162, 163, 164] or middleware/OS-level instrumentation [117, 165, 166, 167] (e.g., Sieve [167] utilizes a kernel module *sysdig* [168] which provides system calls as an event stream containing tracing information about the monitored process to a user application).

Root Cause Analysis. A large body of work [6, 117, 166, 167, 169, 170, 171, 172, 173, 174] provides promising evidence that data-driven diagnostics help detect performance anomalies and analyze root causes. For example, Sieve [167] leverages Granger causality to correlate performance anomaly data series with particular metrics as potential root causes. Pinpoint [166] runs clustering analysis on the Jaccard similarity coefficient to determine the components that are mostly correlated with the failure. Microscope [171] and MicroRCA [173] are both designed to identify abnormal services by constructing service causal graphs that model anomaly propagation and by inferring causes using graph traversal or ranking algorithms [175]. Seer [6] uses deep learning to learn spatial and temporal patterns that translate to SLO violations. However, none of these approaches addresses the dynamic nature of microservice environments (i.e., frequent microservice updates and deployment changes), which require costly model reconstruction or retraining.

Autoscaling Cloud Applications. Current techniques for autoscaling cloud applications can be categorized into four groups [47, 48]: (1) rule-based (commonly offered by cloud providers [130, 131, 132]), (2) time series analysis (regression on resource utilization, performance, and workloads) [101], (3) model-based (e.g., queueing networks) [49, 50, 159, 176, 177], or (4) RL-based. Some approaches combine several techniques. For instance, Auto-pilot [101] combines time series analysis and RL algorithms to scale the number of containers and associated CPU/RAM. Unfortunately, when applied to microservices with large-scale and complex dependencies, independent scaling of each microservice instance results in suboptimal solutions (because of critical path intersection and Insight 2.2 in Section 2.2), and it is difficult to define sub-SLOs for individual instances. Approaches for autoscaling microservices or distributed dataflows [49, 50, 51, 100, 159] make scaling decisions on the number of replicas and/or container size without considering low-level shared-resource interference. ATOM [50] and Microscaler [49] do so by using a combination of queueing network- and heuristic-based approximations. Performance modeling has been proposed for managing scientific workflows [176, 178] but suffers from increasing model complexity, parameter estimation, and inaccurate system assumptions for bursty or dynamic evolving cloud environments and heterogeneous cloud applications. ASFM [100] uses recurrent neural network activity to predict workloads and translates application performance to resources by using linear regression. Streaming and data-processing scalers like DS2 [51] and MIRAS [159] leverage explicit application-level modeling and apply RL to represent the resource-performance mapping of operators and their dependencies.

Cluster Management. The emergence of cloud computing motivates the prevalence of cloud management platforms that provide services such as monitoring, security, fault tolerance, and performance predictability. Examples include Borg [34], Mesos [179], Tarcil [180], Paragon [181], Quasar [182], Morpheus [98], DeepDive [183], and Q-clouds [184]. In FIRM, we do not address the problem of cluster orchestration but focus solely on the resource management task. FIRM can work in conjunction with those cluster management tools to reduce application performance SLO violations.

3.6 SUMMARY

In this chapter, we have described *FIRM*, an ML-based, fine-grained resource management framework that addresses SLO violations and resource underutilization in microservices. FIRM uses a two-level ML model, one for identifying microservices responsible for SLO violations, and the other for mitigation. The combined ML model reduces SLO violations up to 16× while reducing the overall CPU limit by up to 62%. Overall, FIRM enables fast mitigation of SLOs by using efficient resource provisioning, which benefits both cloud service providers and microservice owners. FIRM is open-sourced at https://gitlab.engr.illinois.edu/DEPEND/firm.

CHAPTER 4: DEALING WITH CLOUD MULTI-TENANCY

4.1 INTRODUCTION

In serverless Function-as-a-Service (FaaS) [80, 185, 186], cloud providers handle resource management for each function (e.g., scaling the number of function containers or their resource limits, and scheduling containers to servers) but existing commercial cloud providers provide no performance guarantees such as service-level objectives (SLOs) [187]. Providing performance SLOs has been studied in various aspects and is critical to run latency-critical services on serverless platforms [187, 188, 189, 190, 191, 192]. The problem of managing resources to achieve performance SLOs while maintaining high resource utilization is at its core an intractable NP-hard problem [193, 194]. While the majority of the associated problems are approached using meticulously designed heuristics with extensive application- and system-specific domain-expert-driven tuning, a substantial line of work has recently been focused on learning-based approaches such as reinforcement learning (RL) [1, 16, 63, 65, 66, 159, 194, 195, 196, 197, 198, 199, 200, 201, 202].

As a viable alternative to human-generated heuristics, RL enables an artificial agent to learn the optimal *policy* directly from interaction with the *environment* by observing its *state* and selecting an *action* from the policy. As a result, the environment transitions to the next state, and the agent receives feedback in the form of *rewards*. The goal is to take a sequence of actions that maximizes the expected cumulative rewards in the future. As learning continues, the agent can optimize for a specific workload and adapt to varying conditions (i.e., the *policy-training* stage). After convergence, the learned policy will continue being used by the agent to interact with the environment (i.e., the *policy-serving* stage) [203].

Motivation. Despite recent successes, existing RL-based solutions are all single-agent RL (S-RL) in which one agent manages one or more applications or functions[9] (as shown in Fig. 4.1), paying no attention to any other agents. The standard assumption for an S-RL algorithm is *environment stationarity* [9, 10, 11], which means that the environment is affected only by the agent interacting with it. Therefore, existing S-RL solutions assume that the agent is in an isolated single-tenant environment that contains only the application that the agent manages. In contrast, a serverless FaaS platform in any cloud data center is multi-tenant, and functions from all customers compete for shared resources in a cluster. Multi-tenancy makes the environment *non-stationary* from each agent's own perspective when all agents are jointly being trained. At the training stage, since the state transitions

[9] S-RL-based approaches are application-specific or function-specific.

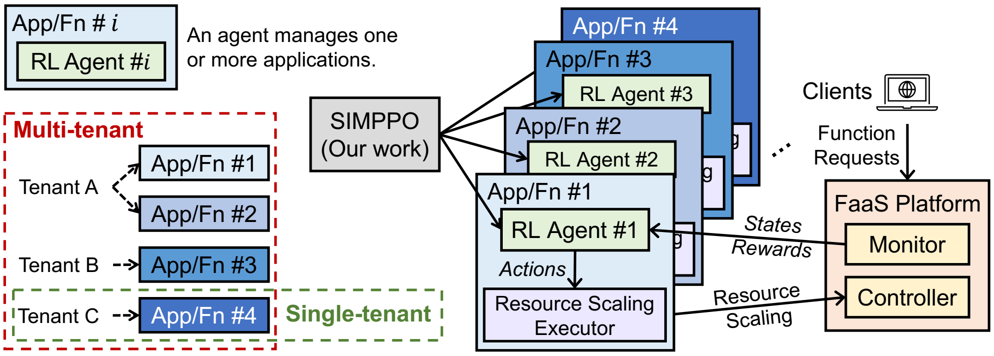


Figure 4.1: Single-agent RL solutions (each agent independently trained and unaware of each other) in single- or multi-tenant serverless environments. scalable and incremental multi-agent RL framework based on Proximal Policy Optimization (SIMPPO) is a multi-agent framework managing all RL agents.

and rewards each agent gets depend on the joint actions of all agents whose policies keep changing in the learning process, each agent enters an endless cycle of adapting to other agents. At the policy-serving stage, an action might be suboptimal when applied, because the underlying environment is no longer the same as the one previously perceived for generating the action. Two examples in Section 2.4 demonstrate such undesirable behaviors caused by environment non-stationarity.

Challenges. We propose a multi-agent RL (MARL) solution that allows multiple agents to coexist in a shared environment. However, solving the training non-convergence problem and achieving policy-serving performance comparable to the performance obtained in single-tenant scenarios further present two main challenges.

(1) *Scalability*: The solution should scale to a large number of functions in a multi-tenant serverless platform. Existing MARL approaches, such as centralized MARL [12, 204] or decentralized MARL with networked agents [205, 206], all suffer scalability issues as the computation complexity of searching in the joint state-action space grows exponentially with the number of agents.

(2) *Adaptive and incremental training*: The proposed solution should adapt to dynamic changes in the environment due to function churns (i.e., add, remove, or update functions) or variations in the number of agents [207, 208]. Existing MARL approaches model agents jointly by optimizing over the Cartesian product of all the agents' action spaces. Consequently, they are computationally inefficient (with exponential complexity), and the whole MARL algorithm must be repeatedly retrained when the number of agents in the environment changes.

Our Work. We design and implement SIMPPO, a **S**calable and **I**ncremental **M**ulti-agent

RL framework based on an RL algorithm, **P**roximal **P**olicy **O**ptimization (PPO) [209]. As illustrated in Fig. 4.1, each agent in SIMPPO still has its own policy trained using PPO and manages an individual function. We use PPO because it provides more stable policy learning, shorter training times, and higher rewards after convergence compared to other RL algorithms (e.g., DDPG [210], DQN [211]). SIMPPO manages all agents to dynamically and continuously maintain the SLO of each function while keeping high utilization efficiency. Each SLO is associated with a function instance specifying a latency threshold for users sending requests to the function instance. In contrast to S-RL, in which each agent is unaware of the other agents and trained in isolation, all agents in SIMPPO are jointly trained to reach convergence. Each agent is allowed to peek into the behavior of the other agents, and changes in other agents' behavior (policy) are handled during training.

To address the incremental training challenge, we designed the MARL model and the neural network architecture of each agent such that all other agents are treated as part of the environment; thus, the model is agnostic to agent sequence order or the number of agents. From each agent's perspective, we created a *virtual* agent that consists of the environment and all the other agents. The many-agent problem is converted to a two-agent problem, so there is no need to reconstruct the neural network, whose structure remains unchanged. In contrast, retraining from scratch for each agent would have been prohibitively costly in both time and resources. Finally, to further shorten the incremental training time, we leveraged neural network parameter sharing [212, 213] between new agents and existing agents that manage the same type of functions.

To address the scalability challenge, we modeled and approximated the collective behavior of the other agents (i.e., the virtual agent) via an *auxiliary global state* distribution. We constructed auxiliary global states by selecting each agent's relevant state and action variables based on domain knowledge and feature engineering to help learn an accurate estimation of the virtual agent. We took the aggregated actions and resource limits from all the other agents to represent the collective resource allocation since we viewed them as part of the environment. We also took the average function performance and resource utilization to indicate how the virtual agent behaves, as the goal is to achieve function SLO performance while maintaining high resource utilization. We removed redundant features that negatively affected RL training because it is hard for the neural network to learn from its unnecessarily large set of inputs. Auxiliary global states are provided to each agent to help it adapt to varying agents in the environment by learning the collective and average behavior of the virtual agent instead of all the other individual agents. Reducing the interaction between one agent and all others to the interaction between one agent and the virtual agent greatly alleviates the scalability issue.

We design SIMPPO as a general framework to support a variety of RL agents that employ online learning algorithms. Although we use serverless resource management as an example for demonstration, SIMPPO can be potentially applied in other RL for systems areas [66] such as load balancing and congestion control. Our approach for using virtual agents draws inspiration from the mean-field theory, which has been successful in economics and physics [214, 215]. Existing theory [216, 217, 218] shows that approximating the collective behavior of all the agents using a single population distribution term such as the average does not lose much optimality in finite multi-agent scenarios, and the approximation error decreases when the number of agents increases.

Contributions. In summary, our main contributions are:

- Single-agent RL formulation and design using the state-of-the-art RL algorithm PPO for serverless resource management (Section 4.3).

- The first quantitative characterization study, to the best of our knowledge, demonstrating that single-agent RL is unable to converge and the policy-serving performance is severely degraded in multi-tenant serverless environments (Section 4.5.2).

- The design of a scalable and incremental MARL framework named SIMPPO that (i) enables multiple RL agents to be trained jointly to convergence and coexist in a multi-tenant serverless environment, and (ii) allows the agents to adapt to dynamic changes in the system, e.g., adding or updating of functions (Section 4.7).

- The implementation and comprehensive evaluation of SIMPPO with real-world workloads that demonstrate scalable and incremental policy training and substantial policy-serving improvements relative to single-agent RL solutions (Section 4.8).

Results. We conducted experiments in which we deployed an open-source serverless FaaS platform, OpenWhisk [219], on our dedicated local cluster and a larger cluster on IBM Cloud. We ran widely used serverless benchmarks [220, 221, 222] driven by arrival rate patterns sampled from both synthetic datasets and public production traces from Azure Functions [186]. We show in our characterization study (Section 4.5) that S-RL trained in isolation (which we use as the *baseline* to assess MARL solutions) (i) improves p99 function latency by at least 1.6× without over-provisioning over a heuristics-based solution in single-tenant cases, and (ii) suffers from 2.2–4.8× higher p99 function latency degradation in multi-tenant cases. An evaluation of SIMPPO shows that it enables all agents' policies to converge when they are jointly trained and that it supports incremental training (in Section 4.8.1). The online policy-serving performance of SIMPPO (in terms of p99 function latency)

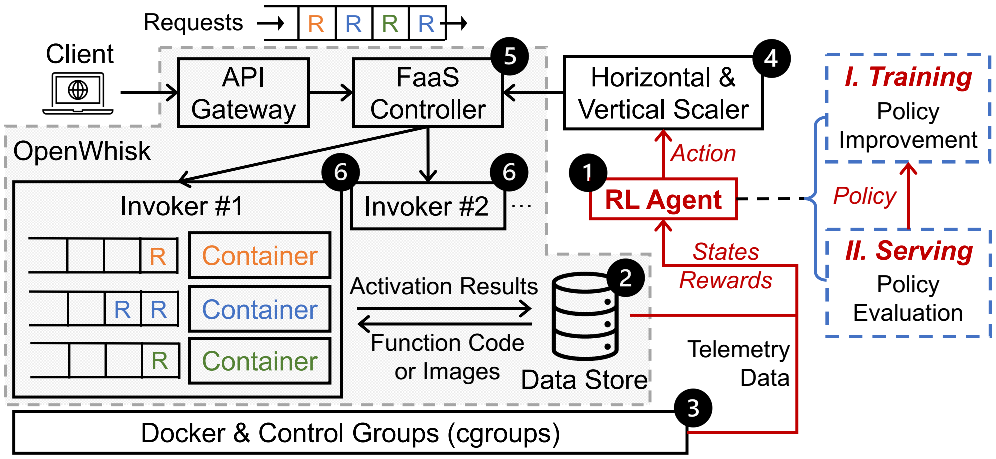


Figure 4.2: Resource management in OpenWhisk [219] with reinforcement learning (RL). At each step, the RL agent perceives system and application conditions from the environment. The measurements are then translated to *state* and *reward* signals that are mapped by the agent to an *action*.

in multi-tenant cases is comparable to the baseline, with <9.2% degradation (in Section 4.8.2); SIMPPO achieves 4.5× improvement compared to S-RL in multi-tenant cases with reasonable resource overhead (in Section 4.8.3). SIMPPO is also scalable to larger numbers of functions and cluster size (in Section 4.8.4).

4.2 BACKGROUND AND MOTIVATION

4.2.1 Serverless Function-as-a-Service

Serverless FaaS is a cloud programming model and architecture wherein customers execute function code snippets without any control over the resources on which the code runs [185]. A serverless FaaS platform runs functions in response to invocations (i.e., requests) from end-users or clients. It consists of a central *controller* and a group of *invokers*. In our study, we chose OpenWhisk [219], a production-grade open-source serverless platform based on Docker containers. Fig. 4.2 shows the architecture of a distributed OpenWhisk platform. The controller (i.e., 5) creates function containers, allocates CPU (`cpu.shares` is used in OpenWhisk [192, 223]) and RAM for each function container, and assigns the containers to invokers (i.e., 6). When requests arrive via the API gateway, the controller distributes the requests to invokers. An invoker executes the function after it receives a request, and the execution results are written to a data store (i.e., 2), which completes an *activation*.

Serverless FaaS workloads, like most cloud data-center services, have service-level objectives (SLOs) defined by each tenant that codify the expected performance [1, 6, 63, 224, 225, 226, 227, 228]. The most common type is a latency SLO, which specifies the acceptable latencies for function requests in the serverless computing context. For example, a latency SLO might specify that 99% of requests have latencies smaller than 100 ms. If a service fails to meet its SLOs, the service provider may risk severe penalties or financial loss. We focus on resource management to meet per-function SLOs while keeping resource utilization at a high level, since low utilization efficiency is undesirable for the cloud provider [179, 182, 228, 229, 230]. The mechanism to convey SLO preferences is also necessary and Henge [231] allows SLOs to be specified by incorporating latency or throughput goals and workload priorities into a user-specifiable utility function.

4.2.2 RL Primer

An RL agent solves a *sequential decision-making problem* (modeled as a Markov decision process or MDP) by interacting with an unknown environment. At each discrete time step t, the agent observes the current *state* of the environment $s_t \in S$, and performs an *action* $a_t \in A$ based on its *policy* $\pi_\theta(s)$ (parameterized by θ), which maps the state space S to the action space A. The agent then observes an *immediate reward* $r_t \in \mathbb{R}$ given by a reward function $r(s_t, a_t)$; the immediate reward represents the loss/gain in transitioning from s_t to s_{t+1} because of action a_t (via `step() function` [203]). The whole sequence of transitions $\{(s_t, a_t, r_t, s_{t+1})\}_{t \geq 0}$ is called an *episode* (or *iteration*). During policy training, the agent's goal is to optimize its policy π_θ so as to maximize the expected *cumulative discounted reward* $\mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t]$ (i.e., the value function) during one episode, starting from a certain initial state s_0, where the expectation is taken over the randomness of state transitions and the agent's policy. The discount factor $\gamma \in (0, 1)$ penalizes the rewards far in the future. After convergence, the learned policy will continue to be used (but not updated) by the agent to interact with the environment during the policy-serving stage.

Two main categories of approaches have been proposed for RL training: value-based methods and policy-based methods [129]. In value-based methods, the agent first learns an estimate of the optimal value function and then approaches the optimal policy by maximizing the estimated value function. In policy-based methods, the agent tries to directly approximate the optimal policy. Both the policy and the value function in RL algorithms are usually implemented using neural networks. We refer readers to [9, 129, 205, 206] for detailed surveys and rigorous derivations of value-based and policy-based RL algorithms.

4.2.3 Related Work

Heuristics-based Resource Management. Various approaches have recently been proposed to address some of the existing challenges in serverless platforms, such as function request scheduling and resource allocation, using carefully designed heuristics [188, 189, 190, 192, 232, 233, 234]. Sequoia [232] is a drop-in front-end for serverless platforms that allows policies to dictate how or where functions should be prioritized, scheduled, and queued. The ideal performance for a specific workload is achieved by carefully designing and evaluating several scheduling algorithms, such as resource-aware scheduling and explicit priority-based scheduling. Atoll [190] is a delay-sensitive serverless framework that exploits a shortest-remaining-slack-first algorithm for scheduling serverless functions. Atoll uses a threshold-based resource scaling method based on queuing delays. ENSURE [192] is another rule-based function resource manager. It allocates $R + c\sqrt{R}$ containers to a function with load R, scales the resources within an invoker based on a latency degradation threshold, and scales the number of invokers based on a memory capacity threshold, tuned per function and per workload.

However, these approaches require recurring human efforts to tune the parameters or choose the appropriate thresholds for each function to achieve optimal performance. For example, threshold-based autoscaling that relies on CPU/memory utilization or latency degradation would be simplistic and inefficient. The parameters need to be reconstructed, tuned, and tested for varying application workloads and infrastructures. Therefore, we focus on RL-based solutions to automatically learn the optimal resource management policies and adapt to frequent changes in serverless function workloads and dynamic cloud environments.

RL-based Resource Management. Lately, RL-based approaches have gained significant momentum toward achieving application SLOs [1, 65, 66, 159, 194, 195, 196, 197, 198, 199, 200, 201, 202]. RL is well-suited for learning resource management policies, as it provides a tight feedback loop for exploring the action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules). Since resource management decisions made for each function are highly repetitive, an abundance of data is generated for training such RL algorithms. RL allows direct learning from the actual workload and operating conditions to understand how the allocation of resources affects application performance. It has been shown that RL with neural networks can express complex system-application environment dynamics and decision-making policies. For instance, FIRM [1] is an RL-based resource management framework for microservices, designed to tackle the under-utilization issue and SLO violations. FIRM uses a two-tier RL model to first identify the microservices that cause SLO violations and then mitigate those violations via dynamic resource reprovisioning. Schuler et al. [195] propose a Q-Learning-based autoscaler that decides on the horizontal concurrency for a serverless function, with the only objective being to minimize the function latency. Zafeiropoulos et al. [196] also apply Q-Learning to threshold-based autoscaling to determine the CPU and memory usage threshold. Both SLO violation and resource utilization are considered in the reward function of each RL agent for a function. FaaSRank [197] is an RL-based serverless function request scheduler that uses PPO to minimize function completion time. However, these works are all S-RL and fail to address non-stationarity in multi-tenant environments (as shown in Section 4.5.2).

Multi-armed bandits such as CloudBandits [235], as a variant of RL (i.e., one-state RL), do not apply in sequential decision problems with state transitions because they require static context with a single decision and reward. Multi-objective Bayesian optimization (BO) [236], on the other hand, is hard to scale to large state space (the known dimension-explosion problem). BO also has a higher latency to find optimal solutions in the search space compared to RL inference.

4.3 SINGLE-AGENT RL FORMULATION

In this section, we present the design for serverless resource management with RL. We call this approach *single-agent RL* or *S-RL* since each RL agent manages one or more specific functions and is unaware of other agents in a shared environment. We first describe the problem formulation as an RL task (Section 4.3.1). We then outline a solution (Section 4.3.2) using a policy-based RL algorithm, Proximal Policy Optimization (PPO) [209].

4.3.1 S-RL Problem Formulation

We model the resource management for each serverless function (by reacting to autoscale after observing a bulk of function executions) as a sequential decision-making problem that can be formulated by the RL framework (illustrated in Fig. 4.2). Since all serverless FaaS platforms have similar controller-worker architectures [186, 190] and function request-serving workflows, for modeling purposes, we chose to use an open-source serverless platform called OpenWhisk [219]. At each step in the sequence, the RL agent (i.e., ❶) monitors system and application conditions from both the OpenWhisk data store (i.e., ❷) and the Linux control groups or cgroups (i.e., ❸). Measurements include function-level performance statistics (i.e., tail latencies on execution time, waiting time, and cold-start time for serving function requests) and system-level resource utilization statistics (e.g., CPU and memory utilization

Table 4.1: State-action space of the RL formulation.

State Space S_t (for single-agent) L_t (for multi-agent)
Function SLO Preservation Ratio ($SP(t)$), Resource Utilization ($RU_{cpu}(t)$, $RU_{mem}(t)$), Function Request Arrival Rate Changes ($AC(t)$), Resource Limits ($RLT_{cpu}(t)$, $RLT_{mem}(t)$), Horizontal Concurrency ($NC(t)$)
Action Space A_t (for both single- and multi-agent)
Vertical Scaling: Resource Limits ($RLT_{cpu}(t), RLT_{mem}(t)$). Horizontal Scaling: Number of Containers ($NC(t)$)
Auxiliary Global State Space G_t (for multi-agent)
Aggregated Resource Limits ($ARLT_{cpu}(t), ARLT_{mem}(t)$), Aggregated Vertical Actions ($AV(t)$) and Horizontal Actions ($AH(t)$), Average SLO Preservation Ratio ($MSP(t)$), Average Resource Utilization ($MRU(t)$)

of function containers). These measured telemetry data are used to define an RL state, which is then mapped to a resource management decision by the RL agent.

Action Space. We consider both vertical and horizontal resource-scaling actions. A vertical-scaling action corresponds to scaling either up or down the `cpu.shares` [223] or the memory limit of a function container, since OpenWhisk's default resource model includes `cpu.shares` and memory limits; both are configurable parameters in all commercial serverless platforms. A horizontal-scaling action in OpenWhisk corresponds to scaling either out or in the function containers, i.e., changing the number of created containers for a function (denoted by NC). The resource limit of each type for a function is initially over-provisioned and later managed by the RL agent. The decision made by the RL agent is then verified and passed by the horizontal and vertical scaler (i.e., ④) to the FaaS controller (i.e., ⑤) and finally changes the resource allocation of the function that the agent manages, and consequently the function performance. As a result, each function instance is deployed in a container with resource limits RLT_{cpu} and RLT_{mem}. The initial limit for each type of resource is over-provisioned before containers are created for a function, and the limit is later controlled by the RL agent. Table 4.1 (the second row) defines the action space which includes available vertical-scaling actions that change RLT_{cpu}, RLT_{mem}, and horizontal-scaling actions that change NC.

State Space. We define the state space based on the five features listed in Table 4.1 (the first row). At each time step t, the average resource utilization $RU(t)$ of a function for each type of resource is retrieved from cgroups (i.e., ③) as telemetry data. The current resource allocations $RLT_{cpu}(t)$, $RLT_{mem}(t)$, and $NC(t)$ are kept as part of the state. In addition, the data store (i.e., ②) also collects function latency composition and request arrival rate. Based on these measurements, the RL agent calculates the remaining two state variables as described below:

(1) *SLO preservation ratio* ($SP(t)$) is defined as `latency_SLO / latency_measured` if there is an SLO violation. The ratio is smaller for more critical SLO violations. Otherwise, $SP(t)$ is set to 1, meaning that there is no SLO violation or no function request.

(2) *Arrival rate change* ($AC(t)$) is defined as $(AR(t)-AR(t-1))$ / $max\{AR(t), AR(t-1)\}$, where $AR(t)$ and $AR(t-1)$ denote the function request arrival rates at the current and previous time steps, respectively. A positive value indicates an increasing request arrival rate and vice versa.

All variables in the state vector are of range $[-1, 1]$ except $RLT(t)$ and $NC(t)$. To facilitate RL training and convergence, we normalized the two variables by setting a predefined resource upper limit $\hat{R}_i$ and a lower limit $\underset{\smile}{R}_i$. For instance, the `cpu.shares` for a container cannot be smaller than 128 or larger than 2048, and the number of containers cannot be smaller than 0 or larger than 1000 (the default maximum concurrency setting in AWS Lambda [237]). If the amount of resources to be vertically scaled reaches the total available amount, then a horizontal-scaling operation is needed.

Reward Function. The goal of the RL agent is, given a time duration T, to learn an optimal policy π_θ that results in fewer SLO violations (i.e., $\max_{\pi_\theta} \sum_{t=0}^{T} SP(t)$) while keeping the resource utilization as high as possible (i.e., $\max_{\pi_\theta} \sum_{t=0}^{T} RU(t)$). Based on both objectives, the reward function is then defined as $r_t = \alpha \cdot SP(t) + (1-\alpha)/2 \cdot (RU_{cpu}(t) + RU_{mem}(t)) + penalty$, where *penalty* is set to -1 in the following cases (and 0 otherwise): (1) Illegal actions such as scaling in/up/down when the number of function containers is zero or scaling beyond the resource limits. Since illegal actions are not executable, an illegal action leads to a self-loop transition from a state to itself with a negative reward. (2) Undesired actions such as frequent oscillating decisions (e.g., scale down and up in two consecutive time steps), which are detected by comparing the actions of the current and last time step. Compared to masking the actions, i.e., excluding the undesired actions from the action space instead of giving negative rewards, our approach leads to faster convergence and is extensible (without modifying the action space).

4.3.2 S-RL Learning Framework

We use a policy-based method, PPO [209], to learn the optimal resource management policy under the RL problem formulation described above. We use PPO because it provides shorter training times and higher rewards after convergence in our setting, compared to other state-of-the-art RL algorithms (e.g., DDPG [210], DQN [211]), while being much simpler to tune. To stabilize the training process, clipping is used to prevent the policy and

Table 4.2: RL training hyperparameters in PPO [209]

Parameter	**Value**
Learning Rate	Actor (3×10^{-4}), Critic (3×10^{-4})
Discount Factor (γ)	0.99
Number of Hidden Layers × Units	Actor (2×64), Critic (2×64)
Mini-batch Size	10 (single-agent), 5 (multi-agent)
Number of SGD Epochs	5
Clip Value (ϵ)	0.2
Entropy Coefficient (β)	0.01
Critic Loss Discount (δ)	0.05
Number of Time Steps (T)	50 (per Episode)
Reward Coefficient (α)	0.3

value functions from changing drastically between training iterations. It has been hypothesized [238] that the smooth policy updates (due to clipping) in PPO can help mitigate the non-stationarity issue in multi-agent RL. Compared to the vanilla policy gradient [239], PPO guarantees an improved policy by specialized clipping in the objective function to prevent the new policy from getting far from the old policy. We implement both the policy π_θ and value function V_ϕ (parameterized by θ and ϕ) of PPO using neural networks. The value function evaluates the expected return of a given state [209]. The policy network maps the state to an action and its parameter θ is updated in the direction suggested by the value function. The hyperparameters used in our PPO implementation are listed in Table 4.2. We set those hyperparameters based on a grid search for the best results. We refer readers to [209] for detailed algorithms of PPO and rigorous derivations and [78] for the pseudo-code. The described S-RL solution is used in our characterization study (Section 4.5) and its performance in single-tenant scenarios is used as the baseline for assessing proposed solutions (as shown in Section 4.6, Section 4.7) in multi-tenant scenarios.

4.4 EXPERIMENTAL METHODOLOGY

OpenWhisk Cluster Setup. We deployed OpenWhisk [219] on five physical nodes in our local cluster, with one master node (which runs the FaaS Controller) and four worker nodes (each of which runs an Invoker), as shown in Fig. 4.2. Each node has a dual-socket Intel Xeon E5-2683 v3 processor with 14 cores per socket and 500 GB memory. All nodes run Ubuntu 18.04 with Linux kernel version 4.15. We also deployed a larger OpenWhisk cluster (20 worker nodes) on IBM Cloud with 22 VMs in us-south-2 to study SIMPPO's scalability. Each node has 8 cores and 16–32 GB RAM, running Ubuntu 20.04. There is no interference from external or background jobs. Docker containers were created on physical nodes in the local cluster and VMs in the IBM Cloud cluster. We disabled memory swapping for the

Table 4.3: Serverless benchmarks [220, 221, 222].

Benchmark	Description
Base64	Encode and decode a string with the Base64 algorithm.
Primes	Find the list of prime numbers less than 10^7.
Markdown2HTML	Render a Base64 uploaded text string as HTML.
Sentiment-Anlys	Generate a sentiment analysis score for the input text.
Image-Resize	Resize the Base64-coded image with new sizes.
HTML-Gen	Generate HTML files from templates.
Uploader	Upload a file from a given URL to Cloud storage.
Compression	Compress given images and upload to Cloud storage.
Image-Inference	Image recognition with a pre-trained ResNet-50 model.
Page-Rank	Calculates the Google PageRank for a specified graph.
Graph-BFT	Traverse the given graph with breadth-first search.
Graph-MST	Generate the minimum spanning tree given a graph.

Docker service. We ran the workload generator [220] and the RL module (either single- or multi-agent) from two separate nodes in the same cluster and used FaaSProfiler [220] to trace requests.

Serverless Benchmarks. We selected benchmarks from widely used open-source FaaS benchmark suites [220, 221, 222] (listed in Table 4.3). These benchmarks include web applications (`HTML-Gen`, `Uploader`), ML-model serving (`Sentiment-Anlys`,`Image-Inference`), multimedia (`Image-Resize, Compression`), scientific functions (`Primes`, `PageRank`, `Graph-BFT`, `Graph-MST`), and utilities (`Base64`, `Markdown2HTML`). These function benchmarks have different runtime behaviors and resource demands in terms of CPU, memory, and I/O utilization. For example, `Image-Resize` and `Image-Inference` are compute-intensive functions; `Base64` and `Markdown2HTML` are memory-intensive functions; `Uploader` and `Compression` are I/O-bound functions; and `Page-Rank` and `Graph-BFT/MST` are data-intensive functions. The functions are written in either Python or Java. SLOs were defined on a per-function basis. In our experiments, we followed the common practice [225] and used the 99th-percentile latency as the SLO latency when running in isolation on the serverless platform. We added a 15% relaxation to the SLO latency to allow fluctuations and measurement errors.

Workloads. In the evaluation, we used both real-world and synthetic function invocation patterns. For real-world workloads, the function invocation patterns are from Azure Functions traces [186] collected over two weeks in 2019. We first constructed a dataset of invocation rates (i.e., RPS) based on the per-minute function invocation count in the traces (the intervals with no invocation will be treated as zero invocation rate). We then uniformly and randomly sampled the invocation rate at each RL step from the constructed dataset. Since our setup of 22 VMs is minimal compared to Azure Functions production setup and to allow the RL agent to explore as many invocation patterns as possible, the invocation

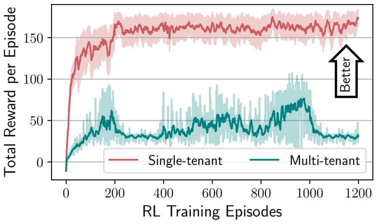


Figure 4.3: Training curves of the single-agent RL (for function `Primes`) in single- and multi-tenant environments.

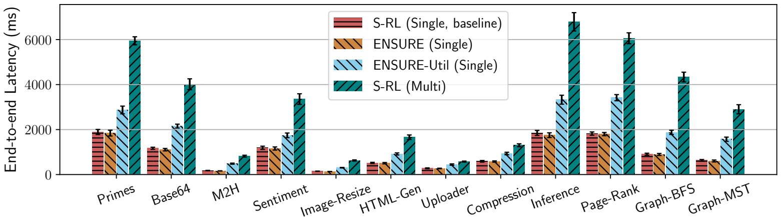


Figure 4.4: Single-agent RL 99th-percentile end-to-end function latency in single- and multi-tenant environments for all function benchmarks in comparison with the heuristics-based approach ENSURE [192] (in single-tenant environments).

pattern of a function in our experiment setup did not follow through with only one function trace but changed to multiple function traces during RL training. For synthetic workloads, we used common patterns [182] indicating flat and fluctuating loads, with a Poisson inter-arrival pattern whose parameter ranges from zero to the maximum value observed in the sampled Azure function traces. The change in request arrival rates was intended to evaluate whether RL agents could adapt to such workload changes.

4.5 SINGLE-AGENT RL CHARACTERIZATION STUDY

4.5.1 S-RL in Single-tenant Cases

We implemented the single-agent RL (S-RL) solution described in Section 4.3, conducted training convergence analysis, and evaluated the function performance and resource utilization compared with those of a state-of-the-art heuristics-based approach, ENSURE [192].

S-RL Policy-training Convergence. To understand the convergence behavior of the S-RL agent in single-tenant environments, we used the workload described in Section 4.4 to train an agent for one function, with no other functions running on the platform. Since RL training proceeds in episodes (iterations), we then analyzed the per-episode reward evolution; Fig. 4.3 (the red curve above) shows the results for the `Primes` function and we found that the agent-training progress is similar across different function benchmarks. We fixed the number of time steps per episode to 50 (i.e., T in Table 4.2). In the initial training stage, the agent policy was unable to mitigate SLO violations. Hence, we terminated the RL exploration early to reset the agent to the initial state and enter the next episode. As the training progressed, the agent improved its resource allocation policy and could mitigate SLO violations in less time. At that point (after around 70 episodes), we linearly increased the number of time steps to let the agent interact with the environment for more time before terminating the exploration and entering the next iteration. The agent's behavior was able to converge after around 300 episodes (ranging from 280 to 350 across all benchmarks).

S-RL Policy-serving Performance. After convergence, we leveraged the trained agent (by using the saved checkpoints at the 1000th episode), and compared it with ENSURE [192], which is a state-of-the-art threshold-based autoscaler implemented on OpenWhisk. We set the parameters and thresholds according to author recommendations [192]. In this comparison, each function controlled by an agent is tested independently. Fig. 4.4 shows the online performance comparison. The S-RL agent is able to keep the CPU utilization at a higher level (around 24% higher than ENSURE, not shown in the figure) and achieves similar end-to-end latency. We found that the reason is that ENSURE over-provisions containers and resources when the workload changes. After increasing the function latency threshold to a higher value (from 15% to 25%), we observed that the performance of the S-RL agent improved over that of ENSURE (labeled "ENSURE-Util") by at least 1.6× with respect to tail latencies at similar CPU utilizations. However, ENSURE does not require any training.

By interacting with dynamic serverless environments under diverse loads and resource allocation scenarios, S-RL agents gradually learn the policy that maximizes the expected rewards and hence outperform heuristics-based approaches. We regard S-RL in single-tenant cases as the baseline for assessing solutions in multi-tenant cases.

4.5.2 S-RL in Multi-tenant Cases

Serverless FaaS platforms are, in essence, multi-tenant, running different functions with various function characteristics, SLOs, and workload patterns from multiple customers [80, 185, 187, 240]. Each function managed by an RL agent competes with other functions on the same platform for limited resources. Function container co-location for higher utilization has made resource contention worse on a cloud serverless platform [80, 187, 220]. The transition from single-tenant to multi-tenant settings introduces new challenges that require a fundamentally different RL algorithm design. Before presenting our multi-agent solution, we first explore the environment non-stationarity issue, and conduct training convergence analysis and policy-serving performance assessment of S-RL agents in multi-tenant environments.

Environment Non-stationarity. The shared environment in a many-agent setting is affected by the actions of all agents; thus, from a single agent's perspective, the environment becomes non-stationary, which breaks the critical stationarity assumption [9, 10, 11] made by S-RL algorithms. In policy training or serving, S-RL requires that the isolated environment is affected only by the agent interacting with it. Under non-stationarity, an agent needs to explore the unknown environment efficiently while keeping in mind that the information it gathers now will soon become outdated, because the other agents are also updating their policies. From a system's perspective, the RL `step()` function (i.e., take an action and receive the next state and reward) can be viewed as a "transaction." An agent interacts with the environment at each time step and commits a transaction that may conflict with other transactions from different agents because of request-response function dependencies or shared-resource contention. The stationarity assumption means that an agent needs to obtain the "lock" to avoid race conditions where the environment ("critical section") is updated by two or more transactions. S-RL fails to provide any synchronization mechanism to address non-stationarity issues.

S-RL Policy-serving Performance Degradation. We assessed the policy-serving performance of the S-RL agent in multi-tenant cases in which each function is in control of an independent S-RL agent trained in isolation (oblivious of the other agents). Fig. 4.4 shows the performance degradation after multiple functions are introduced on the same platform (green bars compared to red bars). In this experiment, we created one function for each benchmark from Table 4.3 and trained one S-RL agent for each function in isolation until convergence. Then, we ran all 12 functions concurrently on OpenWhisk, with each function being managed by its trained S-RL agent. The evaluation results show that the degradation is up to 4.8× (for `Graph-BFS`) and as low as 2.2× (for `Compression`). The reason is that when the agent makes a decision, it is based on the states measured at the current time step; but at the same time, all other agents are also making their resource allocation decisions, which could affect the shared environment. Therefore, the state could have been changed and the estimated value function for an action by the S-RL model is no longer accurate.

S-RL Policy-training Convergence Failure. Multi-tenancy not only affects the online performance during policy serving for S-RL agents trained in isolation, but also leads to problems during training. We trained 12 S-RL agents together; each of them managed one function from the benchmarks listed in Table 4.3. Each S-RL agent was trained independently and did not consider the other agents in the same environment. Everything else was kept the same as before, with the S-RL agent trained in isolation. Fig. 4.3 (the green curve below) shows the per-episode reward evolution for the agent managing function `Primes`. Compared to the training curve of the S-RL agent trained in isolation (in red), the S-RL agent trained in a multi-tenant environment achieved a lower performance (a 55.9% drop in terms of per-episode reward) with higher variance and did not converge in a stable manner within the same training budget. We observed that other function benchmarks also failed to converge within the same training budget (not shown due to page limit).

While S-RL converges in isolated environments and provides a sufficient baseline compared

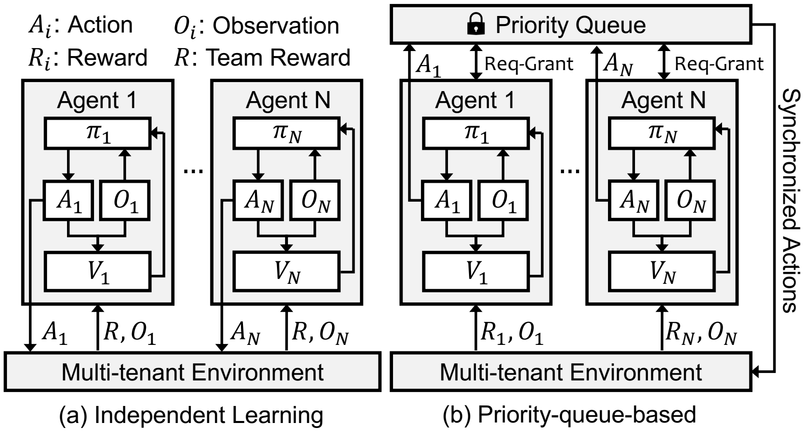


Figure 4.5: RL pipelines for the independent-learning-based approach (a) and priority-queue-based approach (b).

with heuristics, system support for many-agent RL-based resource management that provides both training convergence and comparable policy-serving performance is needed to deal with environment non-stationarity.

4.6 ATTEMPTS TO DEAL WITH NON-STATIONARITY

To deal with non-stationarity in multi-tenant environments, we first designed and implemented two multi-agent RL solutions wherein all agents are jointly trained using independent learning (Section 4.6.1) and a global priority queue (Section 4.6.2). The insights drawn from both solutions lead to our final solution SIMPPO as presented in Section 4.7.

4.6.1 Independent Learning

Our first attempt to deal with non-stationarity is to apply the idea of independent learning [238] to S-RL. In independent learning, each agent learns independently and, by considering only the rewards from all the other agents, perceives the other agents as part of the environment. Despite the loss of theoretical support, independent learners are occasionally shown to achieve desirable results [238]. Therefore we replace the reward function of each RL agent in S-RL with $r_t = \sum_{i=1}^{N} r_t^i / N$, where r_t^i is defined in Section 4.3. We call this the independent learning-based MARL solution *IL-RL*.

Fig. 4.5(a) shows the RL pipeline for the IL-RL training and policy serving. Just as in the S-RL approach, each agent receives the states and rewards from the environment at each step. However, each agent uses the average reward to update its policy network.

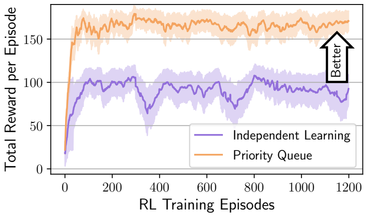


Figure 4.6: Training curve of the IL-RL and CPQ-RL agents (for the `Primes` function) in multi-tenant environments.

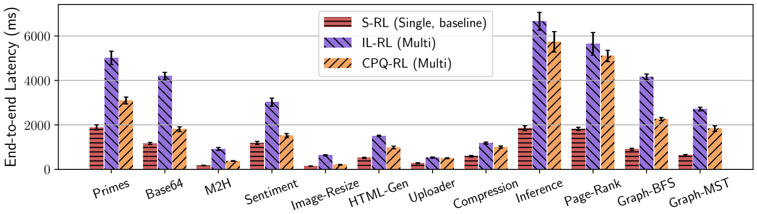


Figure 4.7: The 99th-percentile end-to-end latency comparison for all function benchmarks managed by IL-RL and CPQ-RL agents in multi-tenant environments. The comparison baseline is single-agent RL (S-RL) in single-tenant environments.

IL-RL Policy-training Convergence. To study the training convergence, we created 12 functions (one from each benchmark in Table 4.3), each of which is managed by an IL-RL agent. Fig. 4.6 shows the training curve in this multi-tenant environment. Since all IL-RL agents use the team reward that is the average reward across all agents, Fig. 4.6 (the curve below) shows the evolution of the total reward per episode. Compared to the training curve of S-RL in single-tenant environments (i.e., the baseline, the red curve in Fig. 4.3), the highest reward gained by IL-RL is 38.2% lower than the converged reward achieved in S-RL. However, IL-RL varies less and achieves 1.4× higher reward compared to S-RL trained in multi-tenant environments (the green curve in Fig. 4.3). We conclude that IL-RL fails to provide a convergence guarantee within the same training budget for RL agents in multi-tenant environments.

Insight 4.1*: Simply considering each agent's local observation could work in cases where no extensive coordination with other agents is necessary [238]. However, whenever more complicated coordination is required, such as simultaneously scaling one function out and another function in (as shown in Fig. 2.6), it becomes difficult to explore and learn those joint actions. Without synchronizing the RL state transitions of all agents, one cannot solve the environment non-stationarity issue by considering only the performance of all the other agents.*

4.6.2 Heuristics-based Priority Queue

Since it is hard for agents to independently reason about one another to act intelligently, our second attempt is to use a central coordinator that synchronizes the RL `step()` functions of all agents during both training and policy serving. Fig. 4.5(b) shows an overview of the framework. The coordinator uses a heuristics-based priority queue to determine the execution order of each agent's `step()` function using SLO preservation ratios. Functions with lower ratios (i.e., higher chances of having SLO violations) are served first. We call this central priority queue-based MARL approach *CPQ-RL*.

During CPQ-RL training, the policy executions of the agents are performed one at a time. At the time an RL `step()` function is being executed, no other agent is making any movement. This workflow allows each agent to incrementally improve its policy over time. Intuitively, CPQ-RL provides a training framework wherein each agent gets an isolated sub-environment to make state transitions in the shared multi-tenant environment. The local policy updates of the agents are still distributed and independent from each other.

CPQ-RL Policy-training Convergence. We used the same environment setup described in Section 4.6.1 to train a set of 12 CPQ-RL agents. Fig. 4.6 (the curve above) shows the training curve of the agent that managed function `Primes`. We observed that the training curves of all 12 agents were able to converge. In addition, the total reward per episode of CPQ-RL agents was comparable to the converged values of S-RL agents in single-tenant cases (with <3% difference).

CPQ-RL Policy-serving Performance. We then evaluated the online performance during policy serving by taking the checkpoints of the CPQ-RL agents after the convergence (i.e., at the 1000th episode). Fig. 4.7 shows the end-to-end latency comparison between the baseline (i.e., S-RL in single-tenant environments) and CPQ-RL in multi-tenant environments. Although CPQ-RL provides a training framework that allows each RL agent to learn an optimal policy in an articulated "isolated" environment, there is still 21.8% (`Sentiment-Anlys`) to 67.7% (`Image-Inference`) performance degradation. We found that it was mostly due to the synchronization overhead, since all agents need to communicate with the central coordinator to take the state measurements and make decisions in a sequential manner according to priority. By increasing the priority (setting a tighter SLO), we found that the degradation dropped to less than 35%. Each RL `step()` function (i.e., policy execution) takes around one second. When the number of agents is large (i.e., more than 10), agents with lower priority may suffer from long waiting times (i.e., starvation), which can be unacceptable and lead to cascading SLO violations [6]. Fig. 4.7 also includes the online performance of IL-RL agents, although they fail to converge during policy training. We found that CPQ-RL achieves up to 67.1% performance improvement over IL-RL in terms of p99 function latency, while IL-RL agents perform no better than S-RL agents in multi-tenant environments (which has been shown in Fig. 4.4) with <5% performance difference.

***Insight 4.2**: RL agents with lower priorities suffer from starvation problems in CPQ-RL. A scalable solution relies on simultaneous action generation instead of sequential generation.*

Centralized MARL (such as JAL [12]) could be a potential solution, but it is not scalable and fails to support incremental training (as we describe in Section 4.7.1).

4.7 SIMPPO DESIGN AND IMPLEMENTATION

We describe the design and implementation of SIMPPO in this section, which includes a remodeled resource management problem formulation (in Section 4.7.1) as a multi-agent extension of the Markov decision process (MDP), and a MARL learning framework based on PPO (in Section 4.7.2) under the problem formulation. All agents are jointly trained in SIMPPO by allowing each agent to peek into the behavior of the other agents, which potentially violates the end-to-end arguments [241]. However, the violation is necessary for performance gain [242] as we show in Section 4.5 and Section 4.6 that obeying the end-to-end arguments by handling non-stationarity at each agent's end sacrifices policy-serving performance. Our design choices for the multi-agent model were made to favor scalability and adaptability to agent churn caused by added/removed functions. SIMPPO enables the convergence behavior of all agents during training and provides online policy-serving performance comparable to that of the baseline, i.e., S-RL in single-tenant cases (as shown in Section 4.8).

4.7.1 SIMPPO's MARL Formulation

We extend the MDP formulation for single-agent RL (described in Section 4.3) to a Markov game (also known as a stochastic game [243]) for N agents, each of which controls the resource management for one particular function on the serverless platform. In our formulated Markov game, the state space is defined as the Cartesian product of the state spaces of all S-RL agents (as defined in Section 4.3). After observing the environment state s_t at time t, each agent i takes an action a_t^i based on its policy π_{θ^i} (parameterized by θ^i) and receives a reward r_t^i. The action and reward of each agent are the same as defined in S-RL. The environment state then transitions to a new state that depends on the joint action of all the agents.

A Naive Approach. One can overcome the non-stationarity issue introduced by other agents in the shared environment by using centralized learning (e.g., joint action learners or JAL [12]). In such a centralized approach, the agents are jointly modeled, and a centralized policy for all the agents is trained. The input to this algorithm is the concatenation of the observations of all the agents, and the output is the actions specified to the agents. This approach entirely eliminates the problem of non-stationarity; however, it is computationally inefficient (with exponential complexity). The centralized learner needs to search in the

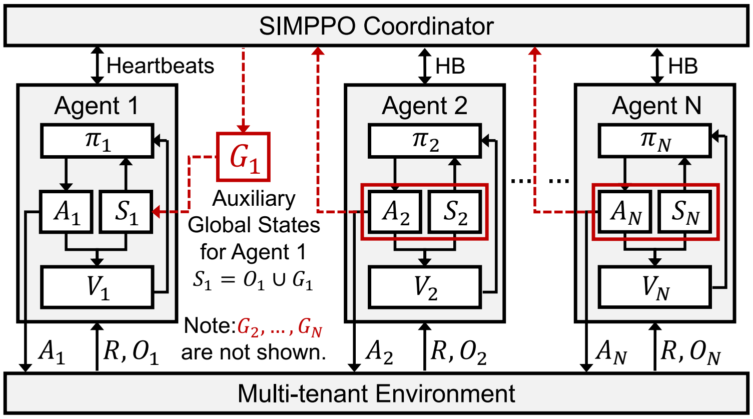


Figure 4.8: Overview of the multi-agent RL model SIMPPO.

joint action space of the size $\prod_{i=1}^{N} |A_i|$ in order to enumerate all possible action combinations, where N is the number of agents and A_i is the individual action space of agent i for $1 \leq i \leq N$. The exponential dependence on N makes the centralized learning approach difficult to scale up beyond a few agents [12]. Therefore, given that there could be tens or even hundreds of function instances on a server [244], we do not proceed further with this centralized approach.

Virtual Agent. The non-stationarity problem leads to significant scalability challenges, since each agent must reason about every other agent's internal state of information. In addition, existing MARL approaches explicitly model each function or agent (e.g., the JAL [12]) and thus the whole MARL algorithm needs to be retrained when there are any changes to the agent group, because the input to the algorithm has been changed. In the typical case where the policy or value functions are parameterized by neural networks, the network structure would also need to be reconstructed. In SIMPPO, from the point of view of an agent i, the main idea is to treat all the other agents as part of the environment by creating a "virtual" agent that represents the environment and all the other agents. Introducing the virtual agent allows SIMPPO to be agnostic to the number of agents and the order of the agent sequence, which enables incremental training. We created *auxiliary global states* (which we will discuss in the next paragraph) by learning the collective and average behavior of the virtual agent instead of each of the other individual agents. Auxiliary global states are provided to each agent to help it adapt to varying agents in the environment. Reducing the interaction between an agent and the others to the interaction between the agent and the virtual agent greatly simplifies the scalability issue.

Value Function Inputs. In SIMPPO's MARL formulation, we resort to using the average to represent the behavior of the other agents (i.e., the virtual agent), and to providing each agent with an auxiliary global state in addition to its individual state. Each agent extracts its local state l_t^i and auxiliary global state g_t^i from the environment state s_t. The local state l_t^i of agent i is from the same state space as in S-RL (Table 4.1 the first row). Both local and auxiliary global states are used as the inputs to the value function, and we find that omitting any local state can be highly detrimental to learning of an optimal resource management policy. In addition, we select a subset of five state-action variables from all variables that can be used to represent the auxiliary global state g_t^i based on domain knowledge. First, aggregated actions and resource limits from all the other agents represent the collective action, since we view them as part of the environment. Second, as the goal is to achieve function SLO performance while maintaining high resource utilization, we select the mean SLO preservation ratio and resource utilization to represent how the other agents behave. Third, we observe that reducing the dimensionality of the value function inputs by removing redundant or repeated features further improves the RL agent performance and reduces training time. For example, a higher arrival rate or a lower horizontal concurrency could have been manifested by a lower SLO preservation ratio. Since each agent only cares about and optimizes its policy based on the reward function, redundant features can negatively affect RL training and make it hard for the neural network to learn to extract the real, useful features. Specifically, g_t^i consists of the following (as listed in the third row in Table 4.1):

(1) *Aggregated CPU limits*: $ARLT^i_{cpu}(t) = \sum_{j \neq i}^{N} RLT^j_{cpu}(t)$

(2) *Aggregated memory limits*: $ARLT^i_{mem}(t) = \sum_{j \neq i}^{N} RLT^j_{mem}(t)$

(3) *Aggregated vertical actions*: $AV^i(t) = \sum_{j \neq i}^{N} \Delta RLT(t)$

(4) *Aggregated horizontal actions*: $AH^i(t) = \sum_{j \neq i}^{N} \Delta NC^j(t)$

(5) *Mean SLO preservation*: $MSP^i(t) = \sum_{j \neq i}^{N} SP^j(t)/(N-1)$

(6) *Mean resource utilization*: $MRU^i(t) = \sum_{j \neq i}^{N} RU^j(t)/(N-1)$

In addition, to measure the volatility of agent performance and behavior, we also include the standard deviation of the selected variables across all the other agents.

Our design draws inspiration from the mean-field theory [216, 217], which has proved to be successful in fields like economics [214] and physics [215]. The underlying principle is to approximate the finite-agent system by summarizing the collective behavior of all agents as a population distribution, which is usually specified as the empirical distribution of the agents' states. Existing theory [216, 217] and our own work [218] show that the approximation error goes down as the number of agents increases because there is less influence from each agent on the overall system.

Rewards. The goal in the MARL setting is (see [205, 206]), given a time duration T, to determine an optimal collection of policies $\pi = \{\pi_{\theta^1}, \pi_{\theta^2}, ..., \pi_{\theta^N}\}$ that result in fewer SLO violations across all functions (i.e., $\max_{\theta^1,\theta^2,...,\theta^N} \sum_{i=1}^{N} \sum_{t=0}^{T} SP^i(t)$) while keeping the average resource utilization of each type as high as possible (i.e., $\max_{\theta^1,\theta^2,...,\theta^N} \sum_{i=1}^{N} \sum_{t=0}^{T} RU^i(t)$). The team-averaged reward function for the MARL setting is then defined as $r_t = \sum_{i=1}^{N} r_t^i/N$, where r_t^i is the reward for agent i as defined in Section 4.3. Our objective is to maximize the expected cumulative discounted reward $\mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t] = \mathbb{E}[\sum_{t=0}^{T} \gamma^t \cdot \sum_{i=1}^{N} r_t^i/N]$.

4.7.2 SIMPPO's Learning Framework

Based on SIMPPO's scalable and incremental MARL formulation in Section 4.7.1, we introduce the learning framework of SIMPPO based on the PPO algorithm that we designed for S-RL (in Section 4.3.2). SIMPPO follows the same algorithmic structure of the PPO algorithm by learning a policy π_{θ^i} and a value network V_{ϕ^i} (parameterized by ϕ^i) for each agent i. Fig. 4.8 shows an overview of the SIMPPO algorithm. We concatenate the auxiliary global state g^i (described in Section 4.7.1) with each agent's local state l^i and feed them as the inputs to the value network V_{ϕ^i}. The auxiliary global state g^i is collected and calculated by the SIMPPO coordinator and sent to each SIMPPO agent. For example in Fig. 4.8, G_1 is sent to Agent 1. The policy network π_{θ^i} is the same as the policy network in S-RL, which maps the states to an action from the same action space. The other extension from S-RL is that the reward for each agent at time t is changed to the average reward across all agents: $r_t = \sum_{i=1}^{N} r_t^i/N$. The RL policy update step of each SIMPPO agent is done in parallel.

Training Data Reuse. PPO uses mini-batch gradient descent to perform several epochs of updates on a batch of training data. We find that the agent achieves optimal performance when the mini-batch size is set to 10 in the single-agent setting. However, in the multi-agent setting, we find that a smaller mini-batch size (i.e., 5) results in a better performance. We attribute this to the negative effect that high data reuse brings in multi-agent settings [245]. The number of epochs implicitly determines the non-stationarity in MARL, as more training epochs will cause larger changes to the agents' policies, which exacerbates the non-stationarity issue. The other hyperparameters are kept the same as in Table 4.2 which are used by the S-RL agents.

Adding and Removing Agents. The SIMPPO coordinator and all SIMPPO agents follow the server-client communication model, and the coordinator is responsible for adding and removing an agent upon request (i.e., when registering or deleting a function). We use a heartbeat-based membership protocol between the SIMPPO coordinator and all SIMPPO agents. When a new function is being added to the serverless platform, a new SIMPPO agent will be initialized to control the resource management of the function. After a new SIMPPO agent is added or an existing function is removed, the auxiliary global state is updated by including or excluding the individual observations from that agent.

Network Parameter Sharing. We leverage neural network parameter sharing [212, 213] between SIMPPO agents that manage the same type of functions to shorten the incremental training time. Based on their sensitivity [147] to the allocation of each type of resource, we categorize the function benchmarks into three coarse-grained categories[10]: CPU-intensive, memory-intensive, and I/O-intensive. One trained RL model is used as the base model for each category of functions. The base model is obtained by training the RL agents that manage all functions that belong to the corresponding category. When a new function is added to the serverless platform, we first categorize the function and initialize the newly created SIMPPO agent with the network parameters from the base model of the same category. The base model can be slowly updated in the background by replacing its network parameters with the existing SIMPPO agents that manage the same type of functions.

4.8 EVALUATION

We evaluated SIMPPO on OpenWhisk driven by widely used serverless benchmarks [220, 221, 222] with both synthetic and real-world workloads from production traces [186], as specified in Section 4.4. Our experiments addressed the following research questions:

- §4.8.1 Does SIMPPO converge and support incremental training? What is the value of auxiliary global states?
- §4.8.2 How does SIMPPO perform in multi-tenant environments compared to single-agent RL in single-tenant environments?
- §4.8.3 What is the resource overhead of SIMPPO?
- §4.8.4 Is SIMPPO scalable with respect to converged rewards, online policy-serving performance, and retraining time?

4.8.1 Incremental Training

During the policy-training stage, we intentionally added and removed a few agents at random from the environment to evaluate the adaptability of the SIMPPO model to agent

[10] We leave fine-grained function classification for better performance to future work.

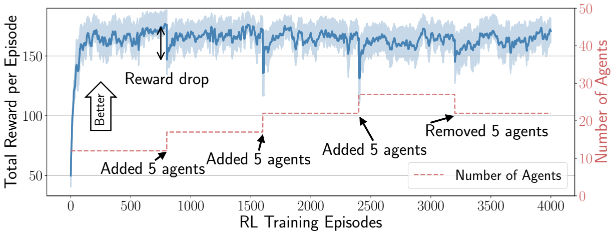


Figure 4.9: Incremental training of the SIMPPO agents.

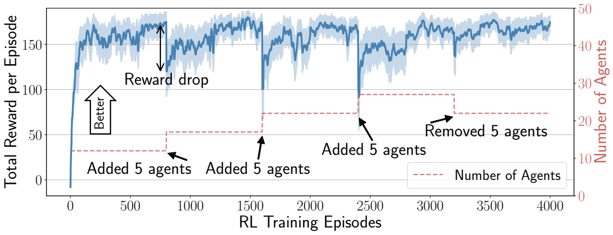


Figure 4.10: Incremental training (no parameter sharing).

updates. Fig. 4.9 shows the training curves of SIMPPO in multi-tenant environments. To start with, we created 12 functions (one from each benchmark in Table 4.3); each function was then managed by an initialized SIMPPO agent. Since all agents used the team reward (i.e., the average reward across all agents), Fig. 4.9 shows the evolution of the average total reward per episode. All agents were able to reach a stable converged policy after around 500 episodes (around 2 hours). Then, at episode 800, we updated the multi-tenant environment by adding five functions (randomly selected from the benchmarks), each of which was managed by a separate SIMPPO agent. As shown in the figure, the total reward per episode dropped by 16.6%; that happened mainly because the five new SIMPPO agents started to learn the optimal policy, which initially leads to low reward values. After around 300 more episodes (around 1.2 hours), the learning curve of the SIMPPO agents was able to converge again. We updated the environment three more times (with one update after every 800 episodes) by either adding five (randomly chosen) new functions or removing five (randomly chosen) existing functions. We observed a similar reward drop and later convergence to about the same level after several hundred training episodes. When we removed five existing functions from the environment, the reward drop (i.e., 11.3%) was not as large as in the previous cases. We attribute the smaller reward drop to the fact that there was no added agent whose reward could start to become randomly lower than that of a trained agent. The team reward still dropped because of the fluctuation of the environment as five functions were removed.

Ablation Study. To study the benefit of neural network parameter sharing brought to incremental training, we implemented a variant of SIMPPO without parameter sharing between any two SIMPPO agents; Fig. 4.10 shows its training curves in the same experimental setup used for the vanilla version of SIMPPO. The benefit of network parameter sharing is manifested in the retraining phase when there is a significant change to the agent group. Compared to SIMPPO without network parameter sharing, the vanilla version has a 20–54.8% smaller reward drop each time functions are added to or removed from the platform. Consequently, the retraining time is reduced by about half (from around 600 episodes) using

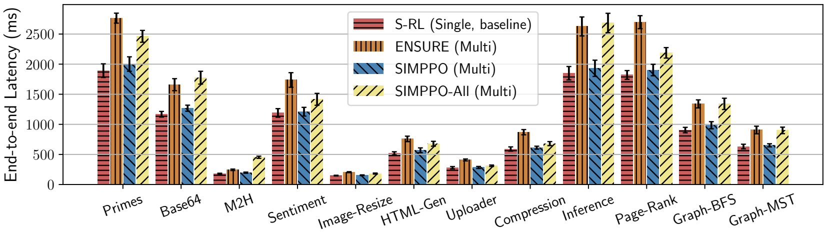


Figure 4.11: The end-to-end latency comparison of functions managed by SIMPPO and the mean-field version of SIMPPO ("SIMPPO-All") agents. The comparison baseline is single-agent RL (S-RL) in single-tenant environments.

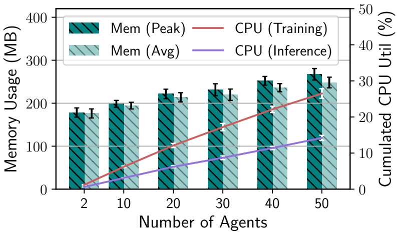


Figure 4.12: Memory usage and CPU utilization overhead analysis of SIMPPO with regard to the number of agents.

the learned base model of each type of function.

4.8.2 Online Policy-serving Performance

We saved the checkpoints at the 3200th episode for all agents in Section 4.8.1 and used those checkpoints to evaluate the online performance during policy serving for all functions together on the serverless platform. We observed that the function performance was similar at different episodes when the agents' behavior converged. At the 3200th episode, there were 27 functions in total (all function benchmarks were used). Fig. 4.11 shows a function performance comparison between SIMPPO in the multi-tenant environment and S-RL trained in a single-tenant environment, i.e., the baseline. We average over all function instances of the same function benchmark. As shown in the figure, SIMPPO is able to provide online performance comparable to that of the baseline, with the performance degradation ranging from 1.8% (for `Sentiment-Anlys`, 1190.2 ms to 1211.5 ms) to 9.2% (for `Markdown2HTML`, 178.8 ms to 196.8 ms). In contrast, the performance degradation for ENSURE in multi-tenant environments ranged from 26.9% to 32.5%, up to 21.4× higher than for SIMPPO (for `Uploader`, SIMPPO improves the degradation from 31.6% to 1.5%). Compared to the single-agent RL trained in multi-tenant environments (as shown in Section 4.5.2 and Fig. 4.4), SIMPPO achieved from 2× (for `Uploader`, 283.4 ms to 576.3 ms) to 4.5× (for `Graph-MST`, 651.8 ms to 2902.6 ms) improvement in terms of the 99th-percentile function latency. In terms of different application categories, we found that SIMPPO performs better for I/O-intensive (1.5–3.6%), then CPU-intensive (3.9–4.2%), and memory-intensive (7.9–9.2%) workloads. We attribute this to better performance predictability of I/O-intensive functions [187], which are less sensitive to memory-bandwidth/CPU contention compared to memory or CPU-intensive workloads.

Ablation Study. To study the importance of feature selection with domain knowledge when constructing the auxiliary global states, we implemented a variant of SIMPPO by replacing the selected auxiliary global states with the average of all individual states from each agent, as suggested in mean-field theory. Fig. 4.11 (in the yellow bars denoting SIMPPO-All) shows its online performance in the same experimental setup used for the vanilla version of SIMPPO. We found that the performance degradation from the single-agent RL baseline in single-tenant settings increased by 2.6× (`HTML-Gen`) to 9.7× (`Graph-MST`) compared to the vanilla version of SIMPPO. The results indicate that our selection of auxiliary global states based on domain knowledge avoids redundant features that increase training time (not shown in this chapter) and negatively affect policy-serving performance because it is hard for the neural network to learn to extract the real, useful features.

4.8.3 Policy Training and Serving Overhead

RL Model Overhead. The actor and critic networks for each SIMPPO agent consist of 4993 parameters (i.e., 28 KB) that are mostly from two neural-network layers with 64 hidden units per layer. Since SIMPPO is implemented in Python with several machine learning libraries such as PyTorch, the imported libraries account for about 166 MB of memory that is shared across all SIMPPO agents. The last part of memory consumption comes from the intermediate data (such as RL state-action transitions and reward vectors) used during training; the size of this memory is proportional to the number of agents and is about 2 MB per agent on average (as shown in Fig. 4.12).

RL Policy-training Overhead. To measure the overhead of RL training for learning an optimal resource management policy, we profile SIMPPO's training process, which proceeds in iterations. The agents all update their model parameters in parallel (using the training approach described in Section 4.7.2). The RL `step()` function takes 240 ± 13 ms, and the actor-critic network parameter update takes an average of 1.34 ± 0.21 s (i.e., five SGD epochs). Thus, in total, training from scratch takes up to around 2 hours, and incremental retraining takes around 0.2–1.8 hours (on Intel Xeon E5-2683). Retraining can be performed infrequently depending on environment stability. We note that since function execution and policy training are asynchronous, the training cost does not directly affect the execution time. However, agents during training are not able to produce optimal decisions although functions can still execute to completion. Most CPU time is spent on network parameter updating, and the overall CPU utilization during training is negligible (i.e., 0.6% per SIMPPO agent, as shown in Fig. 4.12), since the parameter update is performed only once per iteration.

RL Policy-serving Overhead. Mapping of a current state to derive an action requires about 240 ms on average (the same as the RL `step()` function during policy training).

Table 4.4: Scalability analysis of SIMPPO in terms of retraining time in episodes (RT) and reward drop percentage (RDP) after a function churn (i.e., addition or removal of functions).

From	FN Churn	Add New Functions			Remove Functions		
		+2 FNs	+5 FNs	+ 10 FNs	-2 FNs	-5 FNs	-10 FNs
5 FNs	**RDP (%)**[†]	**17.9%**	**20.2%**	38.6%	**9.1%**	—	—
	RT (# Eps)[‡]	**288.5** ± 22	**369.9** ± 18	435.6 ± 24	**129.6** ± 19	—	—
20 FNs	**RDP (%)**	13.3%	16.6%	**42.7%**	9.3%	**10.3%**	**16.2%**
	RT (# Eps)	234.4 ± 21	325.6 ± 20	**455.1** ± 20	133.4 ± 18	**190.5** ± 17	**309.1** ± 20
35 FNs	**RDP (%)**	13.0%	15.4%	31.4%	7.1%	8.8%	14.4%
	RT (# Eps)	216.4 ± 20	284.8 ± 18	315.4 ± 19	120.7 ± 17	167.6 ± 18	268.7 ± 20
50 FNs	**RDP (%)**	9.3%	12.8%	24.2%	5.8%	6.6%	11.0%
	RT (# Eps)	145.8 ± 17	243 ± 16	289.7 ± 21	105.1 ± 14	149.3 ± 16	248.5 ± 18
65 FNs	**RDP (%)**	8.7%	10.3%	21.5%	4.9%	5.9%	11.3%
	RT (# Eps)	150.2 ± 15	208.6 ± 14	247.2 ± 17	82.6 ± 15	126.5 ± 15	208.3 ± 17
80 FNs	**RDP (%)**	6.9%	9.8%	18.5%	3.8%	5.1%	7.8%
	RT (# Eps)	140.0 ± 15	204.4 ± 13	241.8 ± 16	66.4 ± 17	117.5 ± 14	228.6 ± 21
95 FNs	**RDP (%)**	6.0%	8.5%	17.0%	3.2%	**4.7%**	7.4%
	RT (# Eps)	**135.8** ± 13	193.0 ± 12	215.3 ± 15	54.0 ± 15	100.6 ± 12	141.3 ± 17
110 FNs	**RDP (%)**	**5.9%**	**7.7%**	**14.9%**	**3.0%**	4.8%	**6.5%**
	RT (# Eps)	136.1 ± 11	**185.8** ± 13	**196.2** ± 15	**47.2** ± 11	**97.9** ± 13	**132.0** ± 17

[†] RDP = $(R_{before} - R_{after})/R_{before}$ [‡] RT: Num. of RL training episodes for SIMPPO to converge

As the function container resource management is on an independent control plane and asynchronous with the function request scheduling or serving, the RL decision latency does not directly impose any overhead on function execution. The CPU utilization during policy serving is negligible (i.e., 0.3% per SIMPPO agent). With reasonable resource overhead, SIMPPO still improves utilization efficiency.

4.8.4 Scalability Analysis

Finally, we deployed an OpenWhisk cluster with SIMPPO on 22 VMs on IBM Cloud. One VM with 8 cores and 32 GB of memory hosted containers for the controller and other main components, including Nginx and Kafka. One VM with 16 cores and 16 GB of memory hosted all RL agents. Each of the remaining 20 VMs had 8 cores and 16 GB of memory, and hosted an invoker to run the functions in Docker containers. We first confirmed that SIMPPO's training convergence and online policy-serving performance were unchanged compared to those for the local cluster for settings specified in Section 4.8.1 and Section 4.8.2. We then further increased the number of registered functions from 20 to 130 (with the total available memory capacity and cores scaled linearly with the number of functions). We swept over the number of functions to add or remove functions (i.e., 2, 5, or 10 functions) each time function churn happened.

Incremental Training. At each function churn, we recorded the per-episode reward

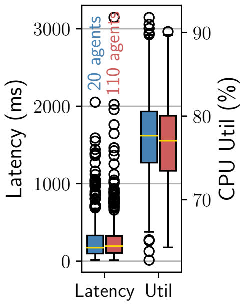


Figure 4.13: Online RL policy serving.

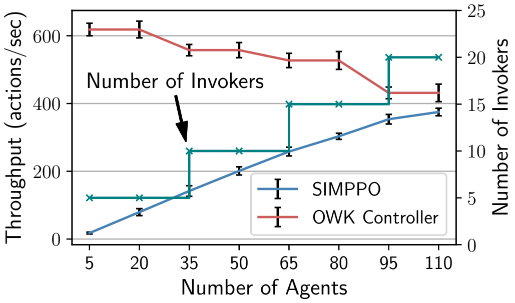


Figure 4.14: RL action throughput.

drop from the previously converged reward to the reward received at the first episode after the churn. To evaluate the effect of agent group size on SIMPPO's incremental training, we also recorded the retraining time required to reach convergence. Table 4.4 shows the reward drop percentage and the retraining time (in terms of the number of training episodes). We observe that as the number of functions increases from 5 to 110, the reward drop percentage first increases and then decreases after the number of functions is greater than 35. When we added 2 or 5 functions at a time, SIMPPO had the highest reward drop (i.e., 17.9% and 20.2%) if there were 5 existing functions; when we added 10 functions, SIMPPO had the highest drop (i.e., 42.7%) if starting at 20 functions. Accordingly, the retraining time has similar trends, as retraining is done until the per-episode reward converges to a stable value (not changing by more than 2%). The most expensive training cost ranged from 196.2 to 455.1 training episodes, on average, for each starting number of functions (about 0.8 to 1.8 hours). Removal of functions results in less reward drop and thus less retraining time for each setup, the same as what we describe in Section 4.8.1. As the number of functions increases to 110, the reward drop percentage and the retraining cost decrease to as little as 3.0% and 47.2 training episodes. We attribute this to the formulation of auxiliary global states for each agent that is used to model and approximate the collective behavior of all the other agents. As the number of agents in the system increases, the disturbance introduced at each function churn gets smaller and the approximation error goes down [216, 217, 218].

Policy-serving Performance. We evaluated the online performance of the learned SIMPPO model when the number of agents was 20 and 110 by taking the model checkpoints after convergence. Fig. 4.13 shows that the distributions of both function end-to-end latency and CPU utilization for the model-serving benchmark `Sentiment-Anlys` are almost the same whether the number of agents is 20 or 110. (Similar trends were observed for other function benchmarks but are not shown in the figure.) The percentage difference of the p99 latency for each function between the 20- and 110-agent settings is smaller than 1.9%.

More rigorously, we ran statistical testing to determine whether the two empirical data distributions are the same (i.e., the null hypothesis). The results show that the two-sided p-values for function latency and CPU utilization are 0.56 and 0.67 (> 0.05) so we cannot reject the null hypothesis. Therefore, we conclude that SIMPPO's online policy-serving performance is scalable in the number of agents.

Throughput. As the number of SIMPPO agents increases, the generated resource-scaling actions per time step grows almost linearly (as shown in Fig. 4.14), with the overhead mainly coming from the calculation of the auxiliary global states. Compared to the throughput that the OpenWhisk controller is able to handle, SIMPPO's throughput demand becomes smaller, but the difference shrinks as the number of invokers increases. In Section 4.9, we discuss a potential solution for scaling beyond the controller's limit.

4.9 DISCUSSION

Performance SLOs. No commercial cloud provider offers SLOs on performance (only availability), which hinders the adoption of latency-critical services on serverless platforms. Though we infer SLOs by profiling applications, a mechanism to convey any SLO preference is needed. We also assume that all user requests sent to the same function correspond to the same SLO. However, the RL policy may not converge when an unachievable SLO is specified. SLO-aware serverless resource management could potentially (1) enable the serverless provider to offer SLO guarantees and change pricing models to be SLO-aware; (2) enable the adoption of latency-sensitive applications from traditional cloud computing to serverless computing.

Scalability Bottleneck. To scale scheduling and resource management, the most widely used approach [33, 34, 190, 246, 247] is to partition the cluster into several system pools and have one controller per system pool. This ensures that a controller does not become a scalability bottleneck. In that scenario, SIMPPO can be applied to make optimal decisions within each system pool.

Compatibility with Non-RL-based Approaches. SIMPPO can be applied to non-RL-based approaches by replacing RL's policy network with their static resource management policy. For example, the most common scaling approach in Kubernetes [248] is threshold-based scaling [249]. For a deployment with a target CPU utilization of 50%, if five pods are currently running and the mean CPU utilization is 75% (i.e., state), the controller will add 3 replicas (i.e., action) to move the pod average closer to the target. High-stakes applications or those with high function churn rates could use non-RL-based approaches and train the RL model in an offline manner [210, 211].

Function Churn Rate. High function update frequency could lead to repeated retraining of the RL model whenever there is a significant change to the environment, although the retraining is less necessary for larger agent group sizes (as shown in Section 4.8.4). The retraining overhead could outweigh the benefits of the learning-based approaches if the churn rate is too high. SIMPPO leverages network parameter sharing to facilitate fast incremental training. In addition, the retraining problem could be alleviated in a private cloud or dedicated clusters for enterprise customers, since the applications evolve more slowly than individual customers [250].

Function Payloads. SIMPPO assumes that the payloads to a function are fixed or result in deterministic execution times (e.g., ML model serving). However, as the payload size increases, execution time might increase, and the function SLO should be redefined.

Function Chains. SIMPPO does not explicitly consider function dependencies. Since our approach is reactive (auto-scaling based on observation), dependencies are indirectly addressed through performance or utilization measurements. To explicitly model dependencies, critical service localization in FIRM [1] could be potentially applied.

Extensibility. SIMPPO could be potentially applied to generic multi-dimensional autoscaling or resource management problems but not in the IaaS domain because each tenant manages resources, and maintains performance-SLOs, but cannot peek into others. To allow more than one objective among a wide range of users, a function with different objectives needs to be turned into different instances each of which is then managed by an agent. The RL reward function also needs to be updated if the objective is not about latency.

Multi-type Virtual Agents. Agents managing functions that have different behaviors (e.g., compute- or memory-intensive) can be represented using separate virtual agents. In addition, different percentile statistics, such as median or P99, can be used to help estimate the collective behavior more accurately. We leave the exploration of multi-type fine-grained virtual agents to future work.

4.10 SUMMARY

This chapter has presented the issues involved in infusing multiple learning-based resource management agents in multi-tenant serverless platforms through a quantitative characterization study. We then propose SIMPPO, a scalable and incremental MARL framework that (i) resolves the many-agent training non-convergence problem while providing online policy-serving performance comparable to that of the baseline (i.e., S-RL in isolation), and (ii) is scalable and adaptive to varying agent groups with reasonable resource overhead.

CHAPTER 5: DEALING WITH CLOUD HETEROGENEITY

5.1 INTRODUCTION

Motivation. Reinforcement learning (RL) has become an active area in machine learning research and is widely used in various systems tasks (e.g., resource scaling [1, 2, 59, 159, 199, 251], power management [16, 75, 252], job scheduling [65, 66, 67, 68, 69, 197], video streaming [70, 253], and congestion control [70, 71, 72, 73, 254]). As a viable alternative to human-generated heuristics, RL automates the repetitive process of heuristics tuning and testing by enabling an *agent* to learn the optimal *policy* directly from interaction with the *environment*.

One example is workload resource autoscaling for meeting application service-level objectives (SLOs) while achieving high resource utilization efficiency [1, 59, 198, 251, 255, 256]. Traditional rule-based approaches [257, 258, 259, 260] configure static upper and lower thresholds offline for certain system metrics (e.g., CPU or memory utilization) or application metrics (e.g., request arrival rate, throughput, or end-to-end latency) so that resources can be scaled accordingly when the measured metrics go above or below the thresholds. Tuning and testing of fine-grained thresholds require significant application/system-specific domain knowledge from experts to achieve optimal resource allocation. Further, repeated parameter tuning for each workload can be labor-intensive, especially for microservice-like applications in large-scale production systems. As different types of services may use different amounts of resources (e.g., CPU and memory) and are sensitive to different kinds of interference and workload spikes, a customized threshold has to be set for a service differently.

RL, on the other hand, is well-suited for learning optimal policies, as it models a systems task (e.g., workload autoscaling) as a sequential decision-making problem and provides a tight feedback loop for exploring the state-action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules) [2, 65]. Integrating RL with those complex systems management tasks in production systems can (1) make full use of the abundant monitoring data on applications and the infrastructure, and (2) automate the process of developing optimal policies while freeing operators from manual workload profiling and parameter tuning/testing. For example, FIRM [1]'s RL agent learns an optimal workload autoscaling policy that adapts to specific application workloads with online telemetry data that alleviates the need for handcrafted heuristics (see Section 5.2.2 for details).

Challenges. However, even as RL is starting to show its strength in the systems and networking domains [1, 2, 16, 59, 65, 66, 67, 68, 69, 70, 71, 72, 73, 75, 159, 197, 199, 251, 253, 254],

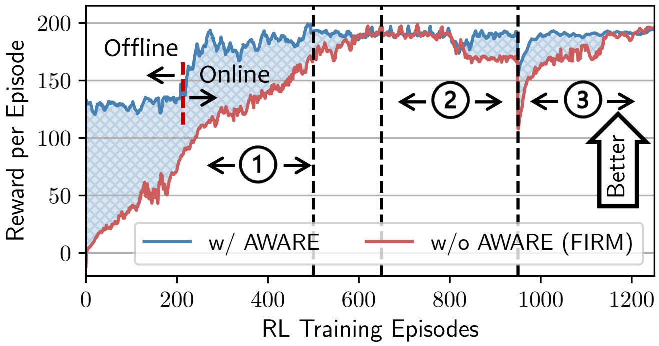


Figure 5.1: RL agent performance when managed by AWARE compared to the baseline (FIRM [1]). Stages ①, ②, and ③ demonstrate the benefit of RL bootstrapping, incremental retraining, and fast adaptation, respectively.

there is still a large gap in directly applying RL advances to real-world production systems due to a series of assumptions that are rarely satisfied in practice. First, a learned RL policy is workload-specific and infrastructure-specific. Retraining is needed to adapt to a new workload or underlying infrastructure in heterogeneous and dynamically evolving (possibly multi-cloud) datacenters [13, 14, 15, 16, 17]. For instance: (1) In SLO-driven resource management, application performance and utilization differ significantly among heterogeneous workloads [1]. Fig. 5.1 stage ③ shows that FIRM's trained RL policy suffers performance degradation and requires substantial retraining. (2) In power management, diverse power consumption and workload sensitivity to core/uncore frequency require separate training of RL policies [16]. (3) In video streaming and network congestion control, different sets of traces have diverse payload characteristics and network environments [70] (e.g., dynamic link bandwidth, delay, and loss rate). Even with transfer learning (TL) [1], nontrivial retraining is needed to adapt to new workloads and environment shifts in each problem domain, which is a critical problem in making RL practical in production. Further, TL requires fine-grained environment clustering to identify the most appropriate model to transfer from, and requires saving one model per cluster.

Second, for the same application and environment, there could be slight changes (e.g., patches and rolling updates), unusual workload patterns not seen before (e.g., due to migration rollout), or traffic jitters. Without timely retraining, the online policy-serving performance of the RL agent fluctuates and leads to undesired degradation (as shown in Fig. 5.1 Stage ②). It is crucial to ensure robust online performance in case of environment or model uncertainty [86, 87].

Third, RL training is through trial and error [1, 65, 69], so worse-than-baseline or suboptimal decisions can be generated, especially at an early stage of training (as shown in

Fig. 5.1 Stage ①). Direct training in the production system leads to suboptimal performance and undesired SLO violations, while training in a simulator and then transitioning to the production system face the problem of poor generalization [88].

A framework that can bring the RL advances to production systems is needed so that (1) the RL model can be trained in a safe and robust manner, (2) the learned RL policy can be adapted to new workloads and altered environments seamlessly without significant retraining, and (3) the online RL model policy-serving performance can be kept stable.

Our Work. We first performed a characterization study of RL in production systems in the task of workload autoscaling. The study focused on the impact of workload change and environment shift regarding (a) RL agent performance degradation or variation and (b) retraining cost. To facilitate the deployment and operation of RL agents in the systems management tasks of a production cloud environment, we introduce an RL model-serving and management framework. As a general framework to support a variety of RL agents for systems management tasks, it can be used by system operators to develop RL-based agents that can be quickly adapted to new environments and achieve stable online policy-serving performance with continuous monitoring and safe bootstrapping (as shown in Fig. 5.1). In the end, system operators can benefit from RL automation of systems management tasks.

- To achieve **fast model adaptation** in each domain or systems task, we leverage meta-learning [261] to model the RL agent as a *base learner* and create a *meta learner* for learning to generalize and adapt to new applications and environment shifts. The base learner discovers policies that generalize across workload variations and intra-environment dynamicity for a pair of application A_i and environment E_j, while the meta learner generalizes across (A_i, E_j) pairs to address inter-environment dynamicity and application heterogeneity. We designed a novel framework that allows the meta learner to learn to generate an *embedding* [261, 262, 263] that projects the application- and system-specific data to a vector space. On this projected vector space, workloads with similar characteristics are projected to closer locations, while those with quite different characteristics are projected to locations far from each other. The embedding is generated by encoding a set of *episodes* from the RL agent's exploration of the environment. Since each episode records a step-by-step interaction of the RL agent with the environment, the time-series episodes naturally encode spatial and temporal characteristics. In the task of workload autoscaling, spatial characteristics correspond to the workload's performance sensitivity to different resource allocations, and temporal characteristics correspond to the time-varying load patterns. The generated embedding is then fed as input to the base learner to adapt to the application and environment shift (from the environments with similar characteristics). With the embedding, fewer retraining iterations are needed for new, previously unseen workloads.

- To achieve **stable online RL policy-serving performance**, we leverage continuous monitoring, and designed a retraining detection and trigger mechanism. An RL agent observes a *state*, performs an *action*, and gets a *reward* at every step in an episode. The time series of states, actions, and rewards in an episode form a *trajectory*. RL trajectories are collected and stored in a time-series database. The most recent rewards are used to calculate the average reward and variation for comparison against user-specified targets. Continuous monitoring ensures that RL model retraining can be triggered or stopped timely so that the RL policy can seamlessly adapt to any environment jitters. We intercept the RL model update logic to enable the switch between RL policy serving and retraining.

- To achieve **safe RL exploration** in the early training stages, we designed an RL bootstrapping module that combines offline and online training. The agent starts with offline training, and a traditional heuristics-based controller (e.g., the Kubernetes Horizontal Pod Autoscaler (HPA) [257] and the Vertical Pod Autoscaler (VPA) [264] in the case of workload autoscaling), is used as the navigator for (online) exploration of the state and action space in the environment. After the RL model is trained to the same level as the heuristics-based controller by comparing rewards, the agent continues to be trained online.

We demonstrate the proposed framework in the task of workload autoscaling on Kubernetes by implementing AWARE (i.e., **A**utomate **W**orkload **A**utoscaling with **RE**inforcement learning). Each RL agent manages a Kubernetes Deployment and configures resources automatically, adjusting both the number of replicas (horizontal scaling) and the CPU/memory limits (vertical scaling) to maintain workload service-level objectives (SLOs) and achieve high resource utilization. To integrate RL agents with Kubernetes, we designed and implemented a multidimensional Pod autoscaler (MPA) system. MPA provides system support for RL-based controllers and translates RL outputs into multidimensional autoscaling actions in a holistic manner by (1) providing an API for RL agents to execute horizontal and vertical scaling decisions on Pod CPU and memory limits, (2) combining vertical and horizontal scaling actions in a single CRD object [265], and (3) providing a user interface for user-defined objective functions for multidimensional autoscaling.

Results. We present a detailed experimental evaluation of AWARE, demonstrating that AWARE significantly improves the practicality of applying RL in production cloud systems

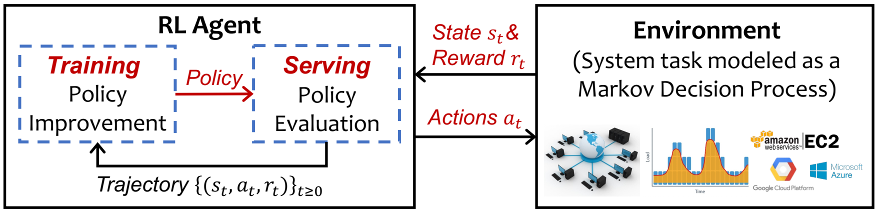


Figure 5.2: An RL agent interacting with an environment modeled as a systems task (e.g., workload autoscaling or congestion control) in the form of a Markov decision process (MDP).

(for workload autoscaling). We first show that the adaptation process of a learned autoscaling policy to new workloads with meta-learning is 5.5× faster than the existing transfer-learning-based approach (Section 5.5.2), and then demonstrate that AWARE provides stable online policy-serving performance with less than 3.6% reward degradation (Section 5.5.3). AWARE's bootstrapping mechanism helps achieve 47.5% and 39.2% higher CPU and memory utilization while reducing SLO violations by a factor of 16.9× during training (Section 5.5.4).

Contributions. In summary, our main contributions are:

- A characterization of RL-based production workload autoscaling and the challenges involved in applying RL in production cloud systems (Section 5.2.3).

- The design of a novel meta-learning-based framework for fast RL model adaptation in workload autoscaling (Section 5.3.2).

- The design of an RL retraining management and bootstrapping mechanism for stable policy-serving performance (Section 5.3.3) and robust RL environment exploration (Section 5.3.4).

- An implementation of the proposed framework in the task of workload autoscaling with MPA, which enables integration of RL agents with Kubernetes (Section 5.4.1).

- A detailed evaluation of AWARE that demonstrates substantial improvements through meta-learning and RL lifecycle management while maintaining workload SLOs and resource utilization (Section 5.5).

5.2 BACKGROUND & CHARACTERIZATION

5.2.1 Reinforcement Learning

In reinforcement learning (RL), an *agent* interacts with an *environment* modeled as a discrete-time Markov decision process (MDP) (as shown in Fig. 5.2). At time step t, the agent perceives a *state* $s_t \in S$ of the environment and takes an *action* $a_t \in A$. The agent receives a *reward* $r_t \in \mathbb{R}$ as feedback on how good the decision is, and at the next time step $t+1$, the environment transitions to a new state s_{t+1}. The whole sequence of transitions $\{(s_t, a_t, r_t)\}_{0 \leq t \leq T}$ is called a *trajectory* or *episode* of length T. The agent's goal is to learn a *policy* π_θ[11] that maximizes the expected cumulative rewards in the future, i.e., $\mathbb{E}[\sum_{t=0}^{T} \gamma^t \cdot r_t]$, where the discount factor $\gamma \in (0,1)$ progressively de-emphasizes future rewards. RL consists of a *policy-training* stage and a *policy-serving* stage [79]. At the policy-training stage, the agent (using an initialized policy) starts with no knowledge about the task and learns by reinforcement and directly interacting with the environment. At the policy-serving stage, the trained policy is used to generate an action based on the current state of the environment, and model parameters are no longer being updated.

5.2.2 Workload Autoscaling with RL

Because of the sequential nature of the decision-making process, RL is well-suited for learning resource management policies, as it provides a tight feedback loop for exploring the state-action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules) [2, 65]. In addition, since the decisions made for workloads are highly repetitive, an abundance of data is generated to train such RL algorithms even with deep neural networks[12]. By directly learning from the actual workload and operating conditions to understand how the allocation of resources affects application performance, the RL agent can optimize for a specific workload and adapt to varying conditions in the learning environment. RL [1, 2, 59, 159, 199, 251] has been shown to automate resource management and outperform heuristics-based approaches in terms of meeting workload SLOs and achieving higher resource utilization.

Specifically, we adopted the design and took the open-source implementation of an RL-based workload autoscaler from FIRM [1], which is the state-of-the-art RL-based autoscaling

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

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

Table 5.1: RL agent state-action space.

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

Table 5.2: RL training hyperparameters.

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

solution, to the best of our knowledge. FIRM uses an actor-critic RL algorithm called DDPG [210].

The RL agent monitors the system- and application-specific measurements and learns how to scale the allocated resources vertically and horizontally. Table 5.1 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, $SP = 1$. An SLO metric can be either request serving latency (e.g., the 99th percentile of the requests are completed in 100ms) or throughput (e.g., request processing rate is no less than 100/s). The reward function is then defined the same as in FIRM [1], $r_t = \alpha \cdot SP_t \cdot |\mathcal{R}| + (1-\alpha) \cdot \sum_{i \in \mathcal{R}} RU_i$, where $\mathcal{R}$ is the set of resources. Table 5.2 lists the hyperparameters tuned for better performance in the experiments. The RL algorithm is trained in an episodic setting. In each episode, the agent manages the autoscaling of the application workload for a fixed period of time (100 RL time steps in our experiments).

5.2.3 Characterization of RL in Production

In the characterization study of FIRM for workload autoscaling, we selected 16 representative production cloud workloads based on a survey of 89 industry use cases of serverless computing applications [266], as serverless workloads are highly dynamic (and thus require autoscaling) and rely on the provider to manage the resources. The selected production workloads include CPU-intensive tasks (e.g., floating-point number computation), image manipulation, text processing, data compression, web serving, ML model serving, and I/O services (e.g., read, write, and streaming). Next, we deployed the selected workloads as Deployments in a five-node Kubernetes cluster in a public cloud and ran an RL-based multi-dimensional autoscaler (i.e., a FIRM agent) with each workload. All nodes run Ubuntu 18.04 with four cores, 16 GB memory, and a 200 GB disk. For RL agent training and inference,

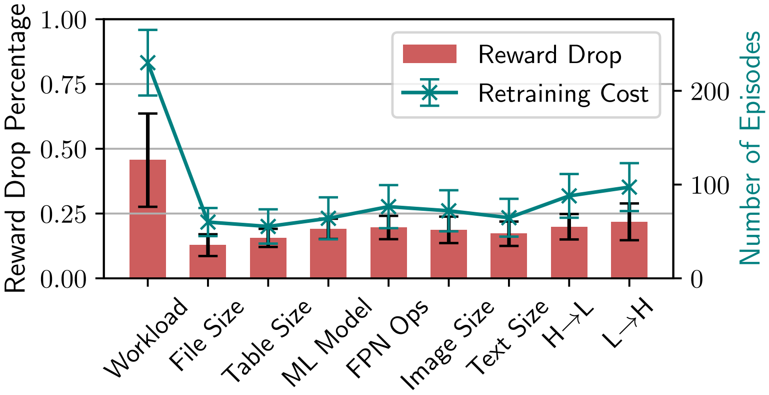


Figure 5.3: Retraining cost of RL models.

we used real-world datacenter traces [267] released by Microsoft Azure, collected over two weeks in 2021.

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

Insight 5.1: Adaptation Retraining Cost. To study the retraining cost of adapting a trained RL policy to new application workloads, we selected each application from the workload pool, trained an RL agent for the application until convergence, and retrained the learned RL policy to serve all the other different applications. We then measured the reward drop after the workload changed and the number of episodes each agent took to retrain to convergence. As shown in Fig. 5.3 (column 1), we observed a 45.6% average per-episode reward drop percentage when the workload had been changed, and retraining to convergence required around 230 episodes (with the model parameter transfer learning used in FIRM [1]).

Insight 5.2: Online Policy-serving Performance Jitters. We introduced seven scenarios to explore the performance instability of RL-based workload autoscaling agents when facing application or service payload size changes and load pattern changes. For I/O services to a backend file system (e.g., AWS S3) and the compression/decompression services, the size of files being read, written, or streaming was changed from [128 KB, 256 KB, 384 KB] to [512 KB, 768 KB, 1024 KB]. For database services, the size of the table being scanned was changed from 1024 items to 10240 items. For floating-point number calculation, the number of operations was changed from 10^8 to 20^8. For image manipulations, the dimension was changed from 40×40 to 160×160. For text processing, the JSON file size was changed from [250 B, 500 B, 1 KB] to [2 KB, 3 KB, 5 KB]. For ML model serving, we changed the matrix multiplication dimension from 50 to 150. For load pattern changes, we divided the Azure workload traces into two parts, one half with a higher daily load ($> 10^5$ per day) and the other half with a lower daily load ($\leq 10^5$ per day).

Table 5.3: Workload performance and utilization efficiency deficit (i.e., the relative difference compared to the rule-based approach) in early-stage RL model training.

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

Fig. 5.3 (columns 2–9) shows the per-episode reward drop percentage and the retraining cost of each scenario. File size changes led to the lowest 12.8% reward drop and around 70 episodes of retraining. We attribute this to I/O-intensive workloads' relatively low sensitivity to CPU/memory allocation, compared to compute- or memory-intensive workloads. Other payload-related changes (i.e., table size, ML model, floating-point number operations, image dimension, and text size) resulted in a 15.6–19.9% reward drop. Load changes from high request arrival rates to low arrival rates (i.e., H→L in Fig. 5.3) and from low rates to high rates (i.e., L→H) resulted in 19.9% and 21.8% reward drops, which required around 98 and 107 episodes of model retraining, respectively.

Insight 5.3: Cost of Early-stage RL Training. As mentioned in Section 5.2.2, RL training proceeds in episodes. When the initialized RL agent starts to learn the optimal policy, especially at an early stage of policy training, the policy might be worse than the baseline heuristics-based approach or even produce undesired actions, such as an oscillating scaling up and down behavior. This is primarily due to the exploration of the state-action space and RL agent learning through trial and error. To study what is lost during policy training, we compared workloads managed by RL agents with the same workloads managed by the rule-based autoscaling approach (i.e., HPA and VPA). We define the early stage of RL training to be the training process from the beginning to the episode at which the RL agent starts to get better than the rule-based approach (which is around 400 episodes in our experiments) because we are interested in the loss due to RL training compared to non-RL-based approaches. We then divided the 400 episodes (in the early training stage) into four segments. For each segment, we calculated the accumulated utilization deficit and SLO violations of the application workloads controlled by the RL agents; the results are shown in Table 5.3. The relative difference in utilization or SLO performance is based on the comparison between the RL agent and the rule-based approach when used to control the same application workloads with the same set of traces.

Results show that RL policies necessarily lead to poor decisions in the early stages of training. In the first 100 episodes, the RL agents inevitably caused more SLO violations than in the other segments (56.1× more than the rule-based approach, which had five SLO violations per 100 episodes). We observe that most SLO violations were due to the under-

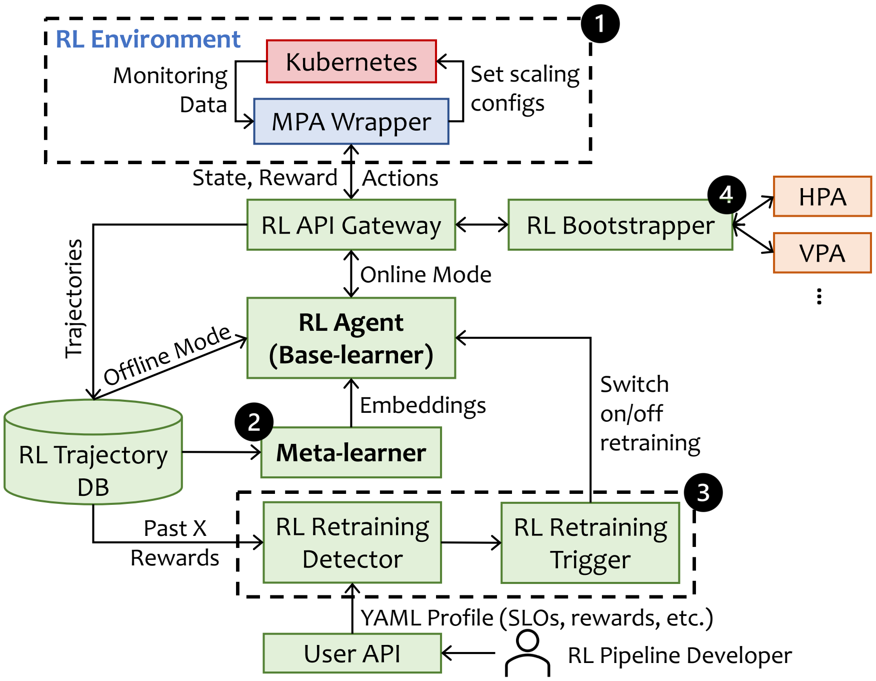


Figure 5.4: Overview of AWARE.

provisioning of resources, so the CPU and memory utilization deficits (32.3% and 28.8% lower, respectively, than for the rule-based approach) were smaller than those in the later segments. In the last three segments, we observe a utilization deficit (i.e., 10–42.9% lower CPU utilization and 7.8–30.5% lower memory utilization) and more SLO violations (i.e., 10.1–22.7×) compared to the rule-based approach.

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

5.3 AWARE DESIGN

5.3.1 Overview

Driven by the insights from Section 5.2.3, we describe the design of AWARE, a framework that supports RL agents for multidimensional Pod autoscaling (MPA) of workloads in production Kubernetes systems. AWARE manages the RL agent lifecycle to deliver stable and robust agent performance. Fig. 5.4 provides an overview of AWARE. We next present a brief summary of each component in this section.

RL Environment. The RL environment (denoted by ① in Fig. 5.4) of AWARE consists of a cluster deployment (e.g., Kubernetes) and an MPA wrapper. The MPA wrapper is designed and implemented as a shim layer that follows an "agent-centric" pattern of request-response interaction advocated by OpenAI Gym [268]. The purpose of the MPA wrapper is to translate measurements and scaling recommendations to and from RL abstractions (i.e., states/rewards and actions), respectively. The communication between the wrapper and the RL agent is through remote procedure calls (RPCs). When the agent steps the environment forward by sending an action to the MPA wrapper through the RPC request, the wrapper translates the received action to vertical and horizontal scaling configurations and applies it to the cluster deployments (e.g., by setting the VPA object [264] and calling the replica re-scaling API). The wrapper gets measurements from the monitoring service in the cluster (e.g., Prometheus [119] in Kubernetes), translates them to RL states and rewards, and sends them back to the agent through the RPC response. The wrapper then waits on the RPC server for the next action request. We describe implementation details in Section 5.4.1.

The framework can also be applied to other systems management tasks (e.g., job scheduling or network congestion control) by replacing the RL environment. Decoupling the RL environment (i.e., the environment wrapper) from the rest of the framework and using the standard OpenAI Gym interface make environment replacement easy [66].

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

RL Agent (Base Learner). The RL agent implements the DDPG RL algorithm (as described in FIRM [1]) and interacts with the RL environment to perform policy training or policy serving (i.e., inference). Since the interface between the RL agent and the MPA wrapper follows the OpenAI Gym standard, different advanced RL algorithms can be used to replace the original RL algorithm DDPG.

Meta Learner. To help adapt to new workloads or environment updates within the problem domain, the meta learner (denoted by ②) selects RL trajectories from the database and generates an embedding that accurately represents the workload running in the environment. RL trajectories are selected per application, and the criteria are based on the reward associated with each trajectory. The embedding is then fed to the base learner (i.e., the RL agent) as part of the input. The RL agent leverages the embedding to adapt (fine-tune) its policy by differentiating heterogeneous workloads and environment updates. See Section 5.3.2 for more details.

RL Retraining Detector and Trigger. At the end of each episode, the RL retraining detector (denoted by ❸) pulls the recent episode rewards gained by the agent from the trajectory database. The mean and standard deviation of the per-episode rewards are calculated and compared to predefined thresholds for performance and variability assessment. If conditions are met, the RL retraining trigger will intercept the inference or training loop of the RL agent to switch retraining on or off, respectively. See Section 5.3.3 for more details.

RL Bootstrapper. The RL bootstrapper (denoted by ❹) determines whether the RL training is online or offline. In the online RL training mode, the RL agent interacts directly with the RL environment. However, offline RL training avoids worse-than-baseline performance or illegal actions in the early stages of RL training, which is desired by production systems. In the offline RL training mode, the RL policy training happens offline based on data collected using a fallback option (i.e., a heuristics-based method), while the RL policy is not used for interacting with the environment. The RL bootstrapper intercepts the request-response path between the RL agent and the RL API gateway and replaces the RL agent with the controller implemented as the fallback option. For instance, in the case of workload autoscaling, the default autoscalers widely used are the traditional rule-based approaches HPA (for horizontal scaling) and VPA (for vertical scaling). Given the states at each time step, corresponding autoscaling actions are then generated based on the HPA and VPA algorithms and sent back to the RL API gateway for execution. The trajectories recorded in the RL trajectory database will be used for the offline RL policy training. See Section 5.3.4 for more details.

5.3.2 Meta Learner

Traditional RL-based resource management approaches [1, 2, 59, 159, 199] require the collection of large amounts of training data samples and retraining (even with transfer learning) to adapt to new environments for (a) updated or previously unseen application workloads or (b) constantly evolving cloud infrastructures [13, 14, 15, 16]. Pure RL-based approaches are no longer tenable in such dynamic cloud environments or even in the context of multi-cloud computing [17]. A novel approach that provides fast model adaptation is needed to make RL practical in production cloud systems.

In AWARE, we leverage meta-learning to reduce the retraining overhead and thus adapt quickly to new environments. In essence, the RL agent is treated as the *base learner* for an individual environment, and a *meta learner* is designed to generate representative *em-*

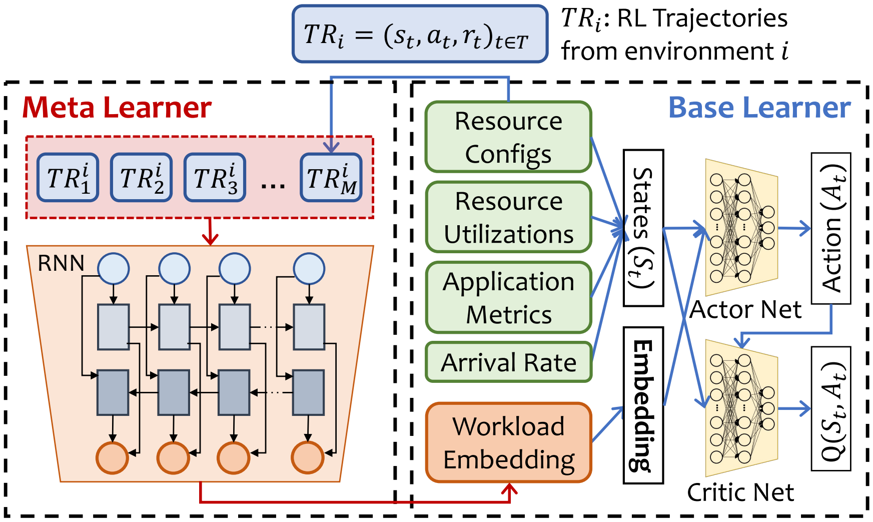


Figure 5.5: Architecture of meta-learning for RL.

beddings that help differentiate environments. We next give a brief primer on meta-learning and the concept of embeddings before presenting AWARE's meta-learning model and an interpretation of embeddings from a systems perspective.

Meta-learning Primer. Meta-learning is known as learning to learn [269]. A good meta-learning model is capable of adapting well or generalizing to new environments that have never been encountered during training time. The adaptation process, essentially a mini-learning session (with limited exposure to the new environment), happens after the meta-learning model training stage. In the meta-learning model training stage, rather than training the learner on a single environment (with the goal of generalizing to unseen "intra-environment" samples from a similar data distribution), a meta learner is trained on a distribution of environments, with the goal of learning a strategy that generalizes to unseen environments (i.e., "inter-environment"). Even without any explicit fine-tuning (i.e., with no gradient back-propagation on trainable variables), the meta-learning model autonomously adjusts internal hidden states to learn [261, 270, 271, 272].

Embedding Techniques. Embeddings map variables to low-dimensional vectors in a way that similar variables are close to each other [262, 263]. Embeddings have been widely used in the area of NLP and software engineering (e.g., word or code embeddings) and can also be applied to dense data to create a meaningful similarity metric. In AWARE, embeddings are used to explicitly represent and differentiate environments, and meta-learning enables learning to generate embeddings.

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

- A Distribution of MDPs (i.e., RL environments): Each MDP corresponds to one agent to which the base learner will adapt. During the training of each agent, the meta learner is exposed to a variety of environments and is trained to adapt to different MDPs. In our case of workload autoscaling, each environment represents a different application workload managed by the base learner, where workloads can have heterogeneous SLOs, payloads, or architecture.

- A Model with Memory: We use a recurrent neural network (RNN) [273, 274, 275] that maintains a high-dimensional hidden state with nonlinear dynamics to acquire, process, and memorize knowledge about the current environment. In an RNN, hidden layers are recurrently used for computation. Compared to memoryless models such as autoregressive models and feed-forward neural networks, RNNs store information in the hidden states for a long time, so they are effective in capturing both spatial and temporal patterns. We did not explicitly use memory augmentation [276] for our RNN meta learner because we found that the features of our application workloads are not as high-dimensional as those of computer vision tasks [276], and the RNN hidden states suffice to provide good representations.

- Meta-learning Algorithm: A meta-learning algorithm learns to update the base learner to optimize for the purpose of adapting quickly to a previously unseen environment [261, 272]. Our novel approach uses an ordinary gradient descent update of RNN with a hidden state reset at a switch of MDPs. As training proceeds, the algorithm learns how to generate an embedding to best represent the environment and differentiate one environment from another.

Integration between Meta Learner and Base Learner. The base learner discovers a rule that generalizes across data points for an (A_i, E_j) pair, while the meta learner generalizes across (A_i, E_j) pairs. Fig. 5.5 illustrates the interaction between the meta learner (❷ in Fig. 5.4) and the base learner. Suppose that each data point used in the training and inference of the RL agent (i.e., a base learner) with the (A_i, E_j) pair i is $\{(s_t, a_t, r_t)\}_{0 \leq t \leq T}$, i.e., one RL trajectory TR_i from the environment i; then, each data point in the meta learner is a bundle of M trajectories from the same environment, i.e., $[TR_1^i, TR_2^i, \ldots, TR_M^i]$. These episodes contain characteristics of the ongoing task that can be used to abstract some specific information about the environment (through <state, action, reward> transition sequences). The meta learner uses a bidirectional RNN [275] to generate an embedding given a sequence of RL trajectories from the same environment (same base learner). Unidirectional RNN has the limitation that it processes inputs in strict temporal order, so the current input has

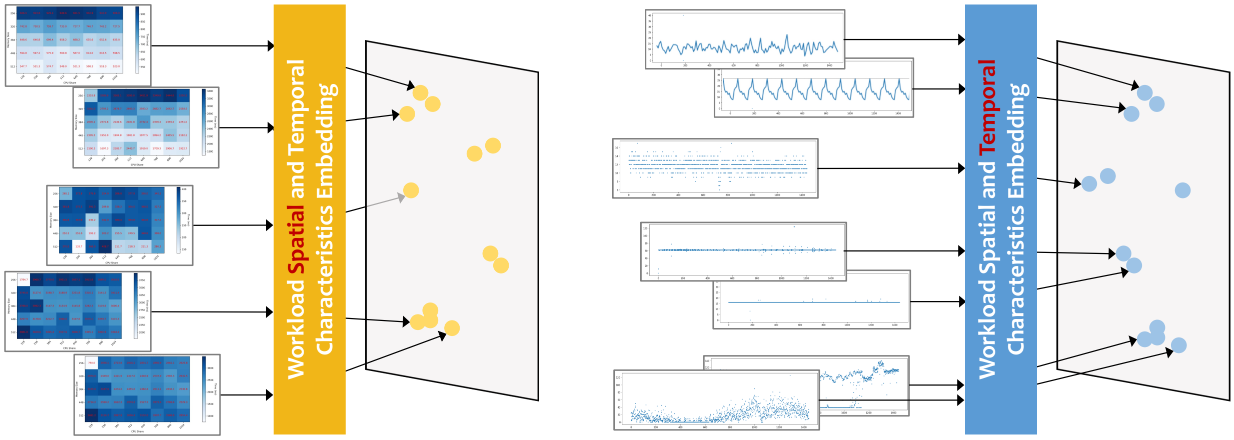

Workload Spatial and Temporal Characteristics Embedding
Workload Spatial and Temporal Characteristics Embedding

Figure 5.6: An example illustrating the idea of workload embedding for encoding spatial and temporal characteristics from a systems perspective.

the context of previous inputs but not the future. Bidirectional RNN, on the other hand, duplicates the RNN processing chain so that the inputs are processed in both forward and backward orders to enable looking into future contexts as well.

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

The output from the bidirectional RNN of the meta learner is an embedding that is used to fingerprint/represent the (A_i, E_j) pair with which the base learner is interacting. As shown in Fig. 5.5, the generated embedding based on past experience (i.e., the episodes previously explored by the base learner) is fed to the base learner as part of the input at each time step. Since we adopted as our base learner the RL design from FIRM [1], which is an actor-critic RL algorithm, the embedding is taken by the actor network.

Interpreting Embeddings from Systems Perspective. The environment-specific embedding is able to differentiate one (A_i, E_j) pair from another and thus guides the base learner to adapt to the new environment. Fig. 5.6 visualizes the key idea of embedding. The spatial and temporal characteristics of the workloads are encoded and mapped onto a low-dimensional latent vector space by the embedding layer. Workloads with similar characteristics are projected to locations that are close to each other on that vector space. For instance, the yellow points (upper figure) indicate that workloads with similar performance sensitivities (to resource allocations) are projected to locations near each other in the embedding vector space. The blue points (lower figure) show that workloads with similar load arrival patterns are projected to adjacent locations in the embedding vector space. By calculating the cosine similarity between any two generated vectors (i.e., embeddings), we can get a monotonic similarity measure. To help understand how generated embeddings can represent spatial and temporal characteristics, we selected RL trajectories from (A_i, E_j) pairs with human-detectable different performance sensitivities or load patterns, and then the plotted embedding projection shows that, indeed, similar workloads are closer to each other when comparing cosine similarities of their embeddings. In Fig. 5.6 (upper), the sensitivity of application performance to different resource allocations is shown in the heatmaps to illustrate the spatial characteristics, with the X-axis being CPU cores and the Y-axis being allocated RAM. Darker colors represent worse performance in terms of application request-serving latency. In Fig. 5.6 (lower), the application load-per-second time series are plotted to represent the temporal characteristics. Again, workloads with similar patterns are projected to adjacent locations in the output vector space.

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

Meta learner Inference. After the meta learner is trained, the meta-learning model is able to adapt the base learner to a new (A_i, E_j) pair that has never been encountered during training. Note that even though the new environment has never been encountered during training, it comes from the same distribution as, or shares similar patterns with, the encountered ones, so that transferring is still possible [261, 270, 271, 272]. The adaptation process only requires limited exposure to the new environment. Therefore, AWARE simply samples RL episodes and runs the meta learner to generate the workload embedding. With the workload embedding, the base learner can be continuously trained to learn the workload autoscaling policy for the new (A_i, E_j) pair. The meta learner model parameters are fixed during the inference stage.

Algorithm 5.1 RL agent lifecycle transition management for bootstrapping and triggering of online retraining. Four status codes `INITIALIZED`, `ONLINE`, `OFFLINE`, and `SERVING` stand for agent-initialized, online training, offline training, and online policy-serving, respectively.

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

5.3.3 Incremental Retraining

When deploying the RL agent in a production system, one needs to ensure that the policy behaves as expected and scales to the workload in production. AWARE leverages continuous monitoring to detect any anomalous behavior and trigger retraining when needed. Alg. 5.1 describes how AWARE's RL retraining module (❸ in Fig. 5.4) manages the lifecycle of the agent and enables incremental retraining at runtime (lines 14–21). The input to the retraining module includes (a) the user profile specifying the configuration, and (b) recent rewards pulled from the RL trajectory database. When the mean and the standard deviation of the recent rewards satisfy the threshold-based condition (i.e., agent performance is bounded to a target value), the agent enters the policy-serving stage; otherwise, the agent enters the policy-training stage. Retraining of the RL agent is done through online interaction with the RL environment. As discussed in Section 5.3.4, non-RL-based approaches (i.e., HPA and VPA) can be used as a fallback option for RL agents when high-stakes applications want to keep the RL agent in the offline mode during retraining.

5.3.4 Bootstrapping

The policy at the early RL training stages could be worse than the baseline approaches. For example, overprovisioning leads to low resource utilization, while under-provisioning results in SLO violations. For production workloads, especially high-stakes applications, such suboptimal actions are not acceptable. In AWARE, an RL bootstrapper (❹ in Fig. 5.4) has been designed to combine offline and online RL training. If the user specifies enabling bootstrapping (as shown in lines 4–9 Alg. 5.1), the offline mode will be turned on first. AWARE will then use Kubernetes HPA [257] (which is a threshold-based approach) for horizontal workload autoscaling, and use Kubernetes VPA [264] (which adjusts resource limits based on history profile) for vertical workload autoscaling. Note that HPA and VPA can also be used as a fallback option for RL when high-stakes applications want to keep the RL agent in the offline mode during retraining, as discussed in Section 5.3.3.

In the offline mode, the RL bootstrapper intercepts the request-response path between the agent and the RL API gateway and replaces the RL agent with the fallback controller to react to the received states and generate actions at each time step. The RL API gateway then takes the received action for execution, and the resulting behavior is the same as when workloads are managed by HPA and VPA. The RL agent samples trajectories from the trajectory database for offline policy training. To overcome extrapolation errors whereby previously unseen state-action pairs are erroneously estimated, we apply a state-conditioned generative model to combine with the critic network for producing previously seen actions [277]. In the online training mode, the agent will then directly interact with the RL environment through the API gateway.

5.4 IMPLEMENTATION

5.4.1 Kubernetes MPA

We propose our own design and implementation of multidimensional Pod autoscaling because the current HPA and VPA controllers are independent of each other and can lead to a large number of tiny Pods [278]. Google MPA [279] is a pre-GA beta version product that offers an integrated solution for HPA and VPA, but it is not open-sourced and does

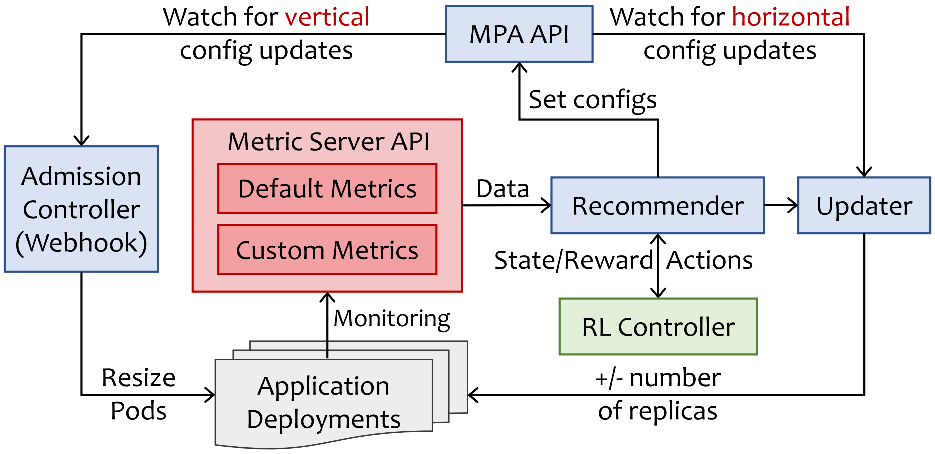


Figure 5.7: MPA design overview and integration with RL.

not support custom recommenders. In AWARE, the MPA framework combines the actions of vertical and horizontal autoscaling but separates the actuation from the controlling algorithms. As shown in Fig. 5.7, there are three controllers (i.e., a recommender, an updater, and an admission controller) and an MPA API (i.e., a CRD object [265]) that connects the autoscaling recommendations to actuation.

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

5.4.2 Integration with RL

The creation of MPA is through declarative YAML files. To integrate MPA with RL agents, one needs to specify a custom recommender to replace the default recommender (HPA+VPA). After an MPA is initialized for the application deployment, an MPA wrapper is created as a shim layer to communicate with the RL agent through RPCs. We follow the "agent-centric" pattern of request-response interaction advocated by OpenAI Gym [268].

The exposed interfaces include (1) `init()` (for initializing the RL environment), (2) `state = reset()` (for resetting the environment at the beginning of each RL episode), and (3) `state, reward = step(action)` (for RL agent stepping). When the MPA wrapper receives an action through the RPC request, it first translates the action to vertical and horizontal scaling configurations and writes to the MPA object. We deploy Prometheus [119], the standard monitoring service in Kubernetes, to export default and custom metrics from the application Deployment. The wrapper then queries the Prometheus service for real-time metrics and translates to RL states and rewards. Finally, the wrapper sends the metrics back to the agent through the RPC response. The MPA wrapper is implemented in Python.

5.4.3 Meta-learning-based RL-serving

AWARE's meta-learning-based RL agent management framework is implemented in Python. Both the base learner (adopted from FIRM [1]) and the meta learner are implemented using PyTorch [139]. The meta learner is essentially a bidirectional two-layer RNN followed by two fully connected layers with the ReLU activation function. We chose the trajectory bundle size to be 20 for the fastest adaptation with the fewest trajectories according to the sensitivity analysis. Each RNN hidden layer consists of 256 neurons, and the fully connected layers consist of 256 and 64 neurons. We chose two layers and an embedding size of 64 because adding more layers and hidden units does not increase performance in our experiments; instead, it slows down training speed significantly. We used the Adam optimizer for parameter updates.

RL trajectories are saved to InfluxDB [280], an open-source time-series database that is built to handle metrics with time-stamped data. Recent rewards, sampled RL trajectories for offline base learner training, and the inputs for embedding generation are all pulled from the trajectory database by using the InfluxDB Python client library.

AWARE provides a simple and declarative user interface for RL pipeline developers, which is consistent with Kubernetes' way of creating and managing objects in the cluster. To specify the targets for the workloads, i.e., resource utilization targets and the application SLO (if there is one), users only need to provide a YAML file following the definition template. Both application latency and throughput SLOs are currently supported. In addition, users can also specify the thresholds for RL rewards and whether or not to enable bootstrapping in the YAML file, which constructs the profile used in Alg. 5.1.

5.5 AWARE EVALUATION

Our experiments addressed the following research questions:

- §5.5.2 Does AWARE provide fast model adaptation to new workloads? What is the value of meta-learning?
- §5.5.3 How does AWARE perform in online policy-serving when workload updates or load changes occur?
- §5.5.4 How does AWARE perform in the early stages of policy training, compared to RL agents without bootstrapping?

5.5.1 Experimental Setup

We implemented an application generator capable of generating a large number of synthetic applications by combining the 16 selected representative production application segments [266] (discussed in Section 5.2.3 as well) based on random sampling with replacement from the segment pool. Each segment represents the smallest granularity of common workloads in cloud datacenters. In addition, each segment has to be associated with its own inputs to simplify load generation (e.g., the image manipulation workloads come with random images). The generator also comes with setup and tear-down scripts for all external services each segment uses (e.g., databases or messaging queues). Overall, we generated 1000 unique applications, deployed them as Deployments in a Kubernetes cluster of 11 two-socket physical nodes, and ran an RL-based multidimensional autoscaler with each application. Each server consists of 56–192 CPU cores and RAM that vary from 500 GB to 1000 GB. Seven of the servers use Intel x86 Xeon E5 processors, and the remaining ones use IBM ppc64 Power8 and Power9 processors.

While it would be impossible to cover all cloud workloads, the selected production workload segments should enable the generation of a large number of synthetic cloud workloads with varying resource consumption profiles. In the future, the number of implemented segments can easily be extended if specific workload profiles are missing. We refer to the open-source artifact for additional details on the generator implementation. With the same datacenter workload traces [267] discussed in Section 5.2.3 with respect to RL agent training and policy-serving, we divided the 1000 generated application pool with the 8:2 ratio. The 800 applications with varied workloads are used to train the meta learner, while the remaining 200 applications are used to evaluate the adaptability. The total runtime is ∼60

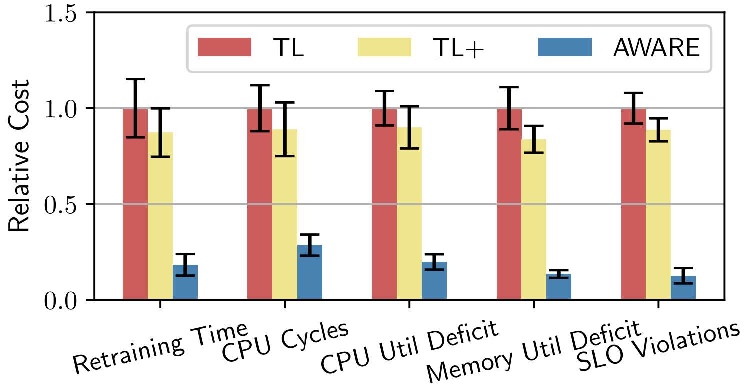


Figure 5.8: RL agent retraining cost and performance comparison of AWARE, transfer learning (TL), and transfer learning with augmented features (TL+).

days, and the meta learner training time is ∼312 hours on an Intel(R) Xeon(R) E5-2695 processor.

The RL formulation and design (in the base learner) are adopted from FIRM [1] (as mentioned in Section 5.2.3). As an end-to-end evaluation, Fig. 5.1 shows that, compared to AWARE, the RL-based autoscaler FIRM by itself suffered from poor performance during the initial training stage (i.e., Stage ①, which demonstrates the benefit of AWARE's bootstrapping mechanism), online policy-serving performance degradation (i.e., Stage ②, which demonstrates the benefit of the online retraining triggering mechanism), and slow adaptation with non-trivial retraining (i.e., Stage ③, which demonstrates the benefit of meta-learning). We then present the evaluation results related to each research question in Section 5.5.2–Section 5.5.4.

5.5.2 Fast Adaptation

To study adaptability to new workloads, we compared AWARE with the existing transfer learning approach. FIRM [1] leverages transfer learning to train an RL agent for a new service based on previous RL experience gained when training the RL agent for a known service. In the transfer-learning-based approach (TL), the model parameters (weights) are shared between the agents managing the known workload and the new workload. We also compared AWARE with a novel approach (TL+) based on transfer learning that includes additional spatial and temporal features in the RL states, since the meta learner in AWARE is trained to output an embedding to represent the spatial and temporal characteristics of the (A_i, E_j) pair. We used the widely used ARIMA model [281] to generate the predicted load for the next time step (i.e., temporal feature) and recorded a table mapping from

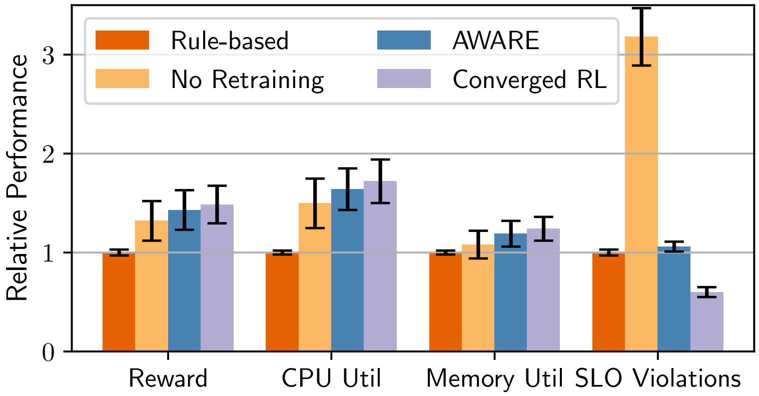


Figure 5.9: RL agent online policy-serving performance comparison of AWARE, no retraining, the rule-based method, and the agent with the converged RL policy. In the comparison of reward and CPU/memory utilization, the higher, the better, while a lower number of SLO violations is better.

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

Fig. 5.8 shows that AWARE adapted 5.5× and 4.6× faster (saved 68–72% CPU cycles) than TL and TL+, respectively. During the adaptation period, TL+ had 4.6× and 6.2× higher CPU and memory utilization deficit compared to AWARE while AWARE reduced SLO violations by 7.1×. TL+ encodes additional spatial and temporal features, but each state is still a stateless snapshot of the running workload. Additional features (i.e., the table and the ARIMA output) greatly increase the state space. Meta-learning, on the other hand, offers a systematic and automated way of learning how to differentiate the workloads well and outputs a low-dimensional embedding to be used by the base learner.

5.5.3 Online Policy-serving

To evaluate the online policy-serving performance when facing workload updates and load changes (described in Section 5.2.3), we compared AWARE with (a) a rule-based approach, (b) an RL agent without continuous monitoring and retraining, and (c) an RL agent with the converged policy. For the rule-based approach with manual scaling, we measured the

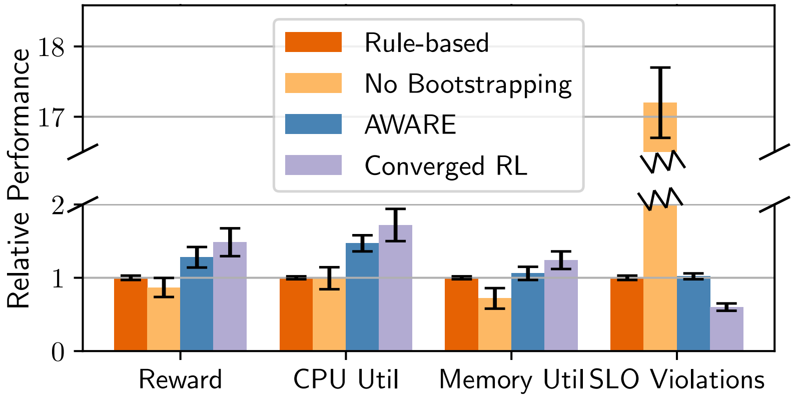


Figure 5.10: RL agent training performance comparison of AWARE, no bootstrapping, the rule-based method, and the agent with the converged RL policy. In the comparison of reward and CPU/memory utilization, the higher, the better, while a lower number of SLO violations is better.

maximum CPU utilization when the SLO was met, and set it as the threshold for HPA. We used the default Auto mode [264] for VPA. We performed the same style of A/B tests 100 times and replaced the MPA recommender with the four approaches. In each A/B test, we randomly selected a workload from the pool, trained the RL agent to convergence, and injected a series of the seven random instability scenarios introduced in Section 5.2.3. We then measured the average reward, CPU/memory utilization, and the number of SLO violations during the time until the agent managed by AWARE converged. Fig. 5.9 shows that AWARE had 9.6% and 14.8% higher CPU and memory utilization, and reduced SLO violations by 3.1× compared to the RL agent without retraining (the second-best approach), resulting in 8.6% higher per-episode reward. Compared to the converged RL policy, AWARE had a 3.6% lower average per-episode reward because we set the retraining threshold to be 5, corresponding to a 2.6% reward degradation. Sensitivity analysis showed that AWARE converged to the no-retraining baseline as the threshold increased while a smaller than 5 threshold led to constant retraining with no policy serving.

5.5.4 Bootstrapping

To study how much bootstrapping helps reduce the cost of early-stage RL training, we compared AWARE with (a) the rule-based approach (same as in Fig. 5.9), (b) an RL agent without bootstrapping, and (c) an RL agent with the converged policy. Since the per-episode reward achieved by the rule-based autoscaler is around 130 (translated from the measured utilization and performance), we set the bootstrapping threshold in AWARE to 130. In

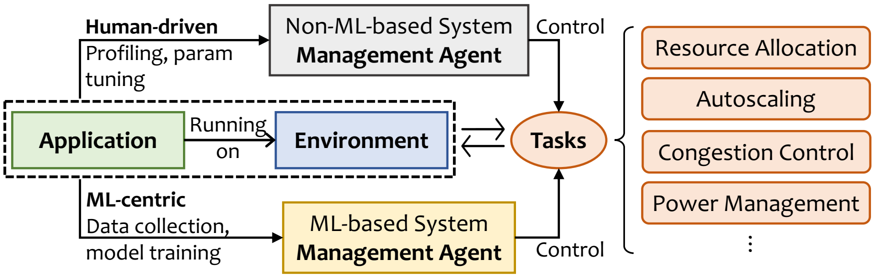


Figure 5.11: An ML-centric cloud platform with a mix of non-ML- and ML-based system management agents.

the sensitivity analysis, we observed that a higher threshold led to endless bootstrapping driven by the rule-based autoscaler (since the measured reward is always lower than the threshold), while the lower the threshold, the more performance degradation AWARE had during its early-stage training. A threshold of 0 basically converges to the learning curve without AWARE bootstrapping (i.e., no offline learning). We performed A/B tests 100 times and replaced the MPA recommender with the four approaches. In each A/B test, we randomly selected a workload from the pool and trained the RL agent to convergence (with or without bootstrapping). We then measured the average reward, CPU utilization, memory utilization, and the number of SLO violations during the time until the RL agent converged. Fig. 5.10 shows that AWARE had 47.5% and 39.2% higher CPU and memory utilization, respectively, and reduced SLO violations by a factor of 16.9× compared to the RL agent without bootstrapping (the second-best approach), resulting in 47.3% higher average per-episode reward before convergence.

5.6 EXTENSION TO GENERAL ML FOR SYSTEMS TASKS

In addition to resource management [1], ML techniques such as supervised learning (SL) and RL have been widely applied in many other system management tasks, e.g., job scheduling [69] and power management [16]. However, from production experience at Microsoft [57], costly model retraining (regarding computation time, energy consumption, and additional hand-made data collection) is a common issue to adapt to previously unseen applications or deployment environments that are constantly introduced in heterogeneous (or even multi-cloud) datacenters [13, 17].

To facilitate efficient ML model adaptation in practice, we extend the idea of meta-learning and 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. The goal is ***not*** to revise existing ML/RL

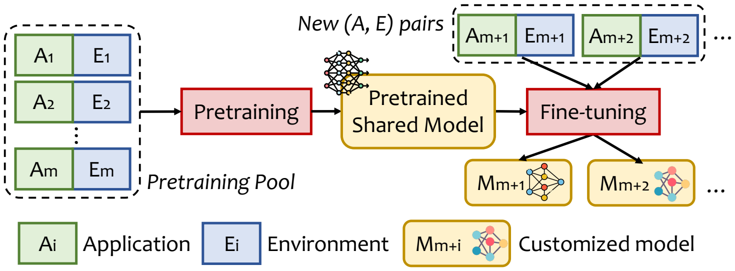


Figure 5.12: Pretrain-finetune workflow in FLASH with the abstraction of application as A_i, environment as E_i, and model as M_i in each system management task.

algorithms or modeling approaches that have been proposed to handle various system tasks but to improve existing model training and adaptation in a *transparent* manner.

Background. FLASH is in line with the vision of an *ML-centric cloud platform* [56] that embeds ML to handle various system management tasks throughout the platform by learning from data and automating management decisions. As shown in Fig. 5.11, 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.

Pretrain-Finetune Paradigm. FLASH is the first general framework that introduces *embedding-based meta-learning* [261, 282] 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) 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 Fig. 5.12. 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*

Table 5.4: FLASH use cases and ML agents developed for cloud system management tasks. See Section 5.6.2 for detailed descriptions.

Use Case	**Application**	**Environment**	**Agent**	**Model**	**Input** / **State** (RL)	**Output** / **Action** (RL)	**Loss** / **Reward** (RL)
Resource Config Search (RCS) [64]	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) [1]	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) [16]	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	$\beta \cdot ((IPS_t - IPS_{t-1}) /IPS_t)_{\Delta freq>0} + (1-\beta) \cdot SP_t$

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 [64] or RL rewards [1]. In autoscaling, the time-varying load patterns (i.e., *temporal* characteristics) are available in RL trajectories [66]. To utilize available datasets and learn to generalize across (A_i, E_j) pairs, we leverage ***meta-learning*** [272], a family of techniques known as "learning to learn" [283], which has demonstrated fast model adaptivity in image classification [282], robotics [261], and cybersecurity [284].

5.6.1 General 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 Fig. 5.13, 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

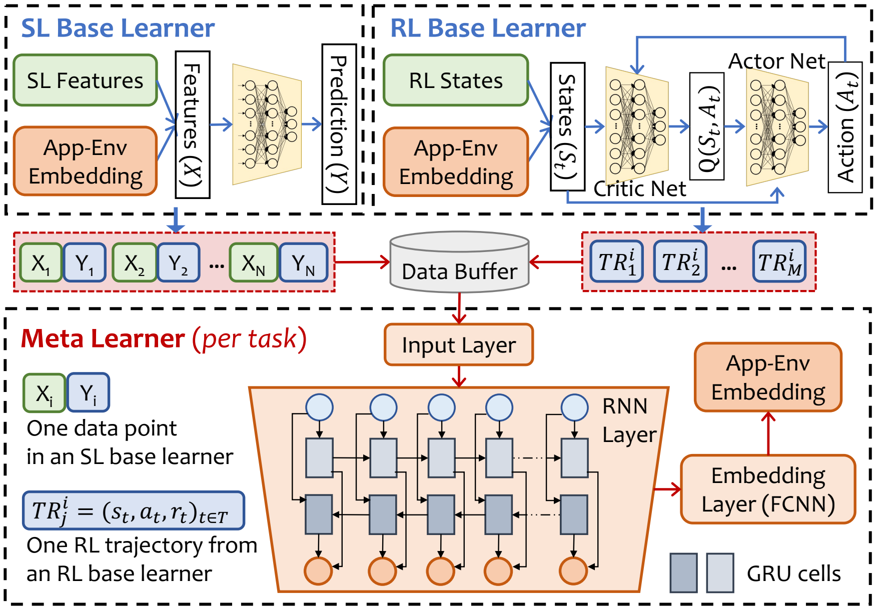


Figure 5.13: FLASH's architecture and integration between the meta learner and base learners (*either* with SL *or* RL).

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) [273, 275] 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 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 [276] 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

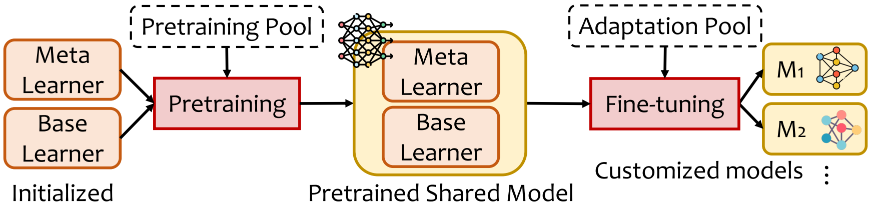


Figure 5.14: FLASH's pre-training and adaptation workflow.

vector) that is used to fingerprint or represent the (A_i, E_j) pair with which the base learner is dealing. As shown in Fig. 5.13, 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.

More implementation details, network architecture, and hyperparameters are presented in Appendix A.1.

Pre-training and Fast Adaptation FLASH is pre-trained on a pool of (A_i, E_j) pairs (i.e., a pretraining pool of applications or environments) and fine-tuned to novel (A_i, E_j) pairs in the adaptation pool, as shown in Fig. 5.14.

- *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 [261]) 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- or 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 [261, 272]. For the pairs with quite dissimilar patterns (identifiable based on the distance in the embedding space as we describe in Section 5.6.3 predictability

```
// 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 5.1: 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.*

analysis), 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.

FLASH Programming Interface FLASH exposes a unified API to ML agent developers that supports both SL and RL as shown in Listing 5.1. 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 Section 5.6.1.

(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 the case of SL or the state vector in the case of 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.

5.6.2 Additional Use Cases

In addition to FIRM, we study two other 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). 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.

Resource Configuration Search. A resource configuration agent (i.e., Sizeless [64]) predicts the performance of a serverless function on various resource configurations. In this case study, we focus on the most recent work, Sizeless [64], 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 Section 2.4, 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 5.4). 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.

CPU Frequency Scaling. A power management agent overclocks CPU cores (by scaling up frequency) only when it benefits the workloads. For instance, SmartOverclock is an intelligent on-node overclocking agent proposed by Microsoft [16]. 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 [285, 286]. 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. However, application heterogeneity leads to quite diverse performance sensitivity to frequency changes, depending on compute or I/O intensity at any given time during the application lifecyle [287]. Therefore, learned frequency scaling policies may not work for a novel set of application workloads.

SmartOverclock uses an RL model, Q-Learning [211]. 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 5.4 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. 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 embedding is retrieved by calling `GetEmbedding()` and then appended to the state vector taken by the Q-Network (in Q-Learning) to complete the value function.

5.6.3 FLASH Evaluation

In each case study, we evaluate (1) the performance degradation of the ML agent with FLASH when testing for new applications/environments, and (2) the efficacy of FLASH's meta learner in fast model adaptation. In addition, we show that a *similarity metric* derived from the generated embeddings can be leveraged to enhance the predictability of adaptation cost.

FLASH-Sizeless Setup. Three datasets are used for evaluation: (1) the original Sizeless dataset [64] consisting of 2000 serverless applications, (2) 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,

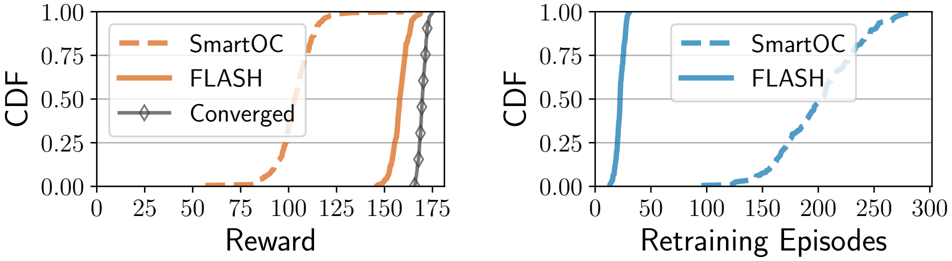
Table 5.5: Sizeless agent prediction error (i.e., MAPE) with and without FLASH.

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%

and (3) the CloudBandit dataset [235] 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 (1) and (2) are used to evaluate the model's robustness across different applications, while dataset (3) is used to evaluate the robustness across different cloud infrastructures (i.e., environments). Details of the dataset are deferred to Appendix A.2.

FLASH-Sizeless Prediction Error without Adaptation. Table 5.5 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 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.

FLASH-SmartOverclock Setup. We leverage the open-source application generators and representative serverless benchmarks from Sizeless to generate 1000 synthetic applica-



(a) Reward Degradation.

(b) Retraining Cost.

Figure 5.15: Comparison of the performance and retraining cost of the SmartOverclock model with FLASH.

tions (as described in Appendix A.4) 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 [267] released by Microsoft Azure, collected over two weeks in 2021. Next, we deploy the selected workloads 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.

In the evaluation, we repeat five experiment runs wherein, in each run, the pretraining of FLASH-SmartOverclock is done on 200 randomly selected applications (i.e., the pretraining pool in Fig. 5.14), 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.

FLASH-SmartOverclock Reward Drop without Adaptation. To evaluate the performance degradation without retraining, we design an autoscaler A/B test where FLASH-SmartOverclock agent and the original SmartOverclock 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 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. Fig. 5.15(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%.

FLASH-SmartOverclock Adaptation Cost. Similar to Section 5.5.2, we evaluate the retraining cost by comparing FLASH with Transfer learning (TL) regarding the number of RL episodes needed to converge. TL retrains an RL agent for a new application based on previous RL experience gained for known applications. The model parameters (weights) are

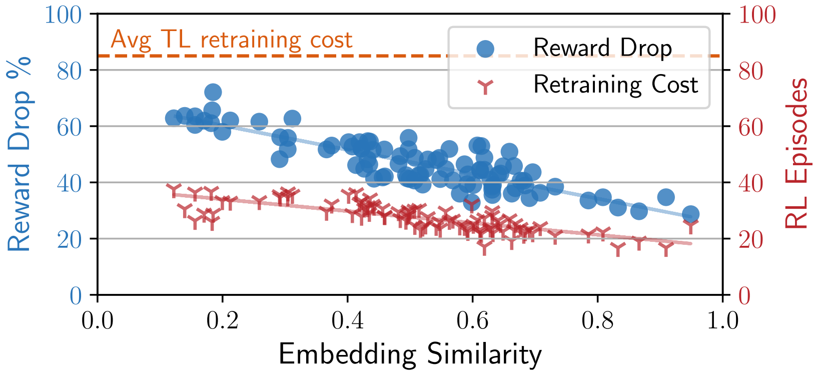


Figure 5.16: Correlation between the embedding similarity and (1) the reward drop percentage without retraining (in blue) and (2) the retaining cost with FLASH (in red).

shared between the agents managing the known workload and the new workload. Fig. 5.15(b) shows the CDF of the RL retraining cost and, on average, FLASH-SmartOverclock adapts 9.2× faster than TL.

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 (1) performance drop and (2) 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. Fig. 5.16 illustrates the correlation between embedding similarity and (1) the reward drop percentage (in blue) and (2) 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.

5.7 RELATED WORK

RL Training and Model-serving Frameworks. Ray [79] is an open-source distributed execution framework that facilitates RL model training and serving by making it easy to scale an RL application and schedule distributed runs to efficiently use all resources (i.e., CPU, memory, or GPU) available in a cluster. Amazon SageMaker [288] uses the Ray RLlib library that builds on the Ray framework to train RL policies. SageMaker also provides cloud services that help build and deploy ML models (e.g., data processing and model evaluation). RLzoo [289] is an RL library that aims to make the development of RL agents efficient by providing high-level yet flexible APIs for prototyping RL agents. RLzoo also allows users to import a wide range of RL agents and easily compare their performance. Park [66] provides 12 representative RL environments in the field of systems and networking (e.g., job scheduling) for developing and evaluating RL algorithms. Genet [70] is an RL training framework for learning better network adaptation policies. Genet leverages curriculum learning [290], which aims to sequence tasks to achieve the best performance on a specific final task instead of quickly adapting to a new task within a small number of gradient descent steps.

RL in Production. Panzer *et al.* [291] provide a survey of existing RL applications in production system domains, including resource scheduling. They summarize the implementation challenges and generalizability of simulation-trained RL models. SOL [16] is an extensible framework for developing ML/RL-based controllers for tasks such as core frequency scaling. SOL is complementary to AWARE, which can further guarantee that the RL agent operates safely under various realistic issues, including bad data and external interference like resource unavailability. SIMPPO [2, 218] provides a scalable framework based on the mean-field theory that enables multiple RL agents to coexist in a shared multi-tenant environment. Autopilot [198] is a workload autoscaler used at Google that leverages multi-armed bandits (i.e., the simplest version of RL) to choose a variant of the sliding window algorithms that historically would have resulted in the best performance for each job. In its essence, it is still a heuristic mechanism and has been shown [59] to suffer from poor system stability because of inaccurate estimation of horizontal concurrency; it can also result in a large number of tiny Pods [278] due to the independence between horizontal and vertical scaling.

Meta-learning for Systems. ResTune [292] leverages meta-learning to optimize hyperparameters that boost Bayesian optimization on database performance tuning knobs. Xue et al. [293] 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 [282]. Other examples of meta-learning model parameter initialization include PSO [294] and DMRO [295], 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 et al. ([296]) 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 [297] and microservices [298]. Meta-learning can be complementary by learning a good representation that can generalize well to help with zero/few-shot learning [299].

Curriculum Learning for RL. We consider curriculum learning [300] (e.g., Genet [70]) 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 [300], meta-learning provides theoretically proved generalizability across tasks.

5.8 DISCUSSION AND FUTURE CHALLENGES

Extension to Other System Domains. AWARE is a general and extensible framework that can be applied to other systems management tasks (e.g., congestion control or job scheduling). To apply it to a new domain, one needs to (1) replace the RL environment by implementing the provided environment wrapper interface; and (2) provide a default non-RL-based agent for the RL bootstrapper. In FLASH, we demonstrate the use of meta-learning in resource configuration search and CPU frequency scaling. We leave the study of the performance in other system domains to the future.

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 [301]). 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 [69] or autoscaling [187]).

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) [84, 302] and Monarch [303] are profiling infrastructures for datacenters, providing performance insights for machines and cloud applications. Cluster managers such as Borg [304] 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 [84].

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). 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 5.4) 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 [305].

Out-of-distribution Workloads. AWARE provides the opportunity to quickly customize the model to specific workloads. However, out-of-distribution (A_i, E_j) pairs still require training because meta-learning assumes that all pairs, including the unseen cases, are inherently within the learned distribution [272] (e.g., in terms of service request arrival patterns or sensitivity to resource allocation). Given the diversity of workloads in the cloud datacenter (used in the training dataset), the meta learner and the shared base learner can be continuously trained, and out-of-distribution cases are covered eventually. Meanwhile, with offline RL training, users can still benefit from the heuristics-based solution used as the fallback option. One limitation of our experiment was that the generated applications might not have covered all possible cloud workloads. However, application segments can easily be extended in the synthetic application generator if specific workload profiles are missing (Section 5.5.1).

On-policy RL Algorithms. When RL agents are being bootstrapped at the initial stage, off-policy RL agents (such as DDPG [1, 210] and DQN [59, 159]) can be trained directly using the collected RL trajectories. However, on-policy RL agents (such as PPO [2]) require trajectories generated from their own policy. One potential way to train on-policy RL agents offline would be to build a simulator based on the collected trajectories, which would essentially map resource allocation to workload performance and system metrics. A balanced experience replay scheme [306] could potentially be applied for locating near-on-policy samples from the simulator constructed based on the offline dataset. Instead of drawing trajectories from the trajectory database (as in Section 5.3.4), the RL base learner can interact with the simulator for bootstrapping.

5.9 SUMMARY

This chapter has presented the challenges of applying RL in workload autoscaling and other tasks in production cloud platforms. We then proposed a general and extensible framework for deploying and managing RL agents in production systems. To demonstrate the framework, we implemented AWARE for automating RL-based workload autoscaling in Kubernetes and experimentally showed (a) the benefits of leveraging meta-learning for fast model adaptation, and (b) how the design of AWARE ensures the stable and robust online performance of RL models. AWARE is open-sourced at https://gitlab.engr.illinois.edu/DEPEND/aware.

Building upon the meta-learning idea in AWARE, we also explored the general 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.

CHAPTER 6: LEARNING DEEP LEARNING MODEL SERVING POLICIES

6.1 INTRODUCTION

Over the past few years, increasingly capable large models have been developed for everything from recommendations to text or image generation. Pre-trained deep learning (DL) models make it easy for developers to develop and deploy new models with lightweight fine-tuning, few-shot learning, or even prompting [18]. As a result, model serving (i.e., inference) has become an essential workload in modern cloud systems.

However, serving deep learning models at scale puts a pressing need to improve ***power efficiency***, i.e., to reduce energy consumption while maintaining model-serving performance requirements. A recent study from Google attributed 60% of its ML energy use to inference [307]. AWS estimates inference to make up 90% of total ML cloud computing demand [308]. Unfortunately, existing model-serving systems focus on either performance (i.e., latency and throughput) [224, 226, 309, 310, 311, 312], model prediction accuracy [313, 314, 315], or LLM optimization [316, 317, 318, 319], but are not power-aware. In this chapter, we target ***power efficiency*** in DL model serving.

Challenges. We could potentially leverage GPU Dynamic Voltage and Frequency Scaling (DVFS) techniques that have been widely used during GPU training [320, 321, 322, 323] to save energy. However, dynamic GPU frequency scaling on modern deep learning model-serving workloads poses three main challenges due to their unique characteristics.

[C1] First, it is challenging to determine the optimal GPU frequency that meets performance requirements while maximizing power efficiency. Since low latency in inference is critical, model-serving workloads are typically associated with service-level objectives (SLOs) [310, 311, 312] on end-to-end latency. To meet stringent SLOs, contemporary serving systems either fix the GPU frequency at the default setting or rely on DVFS to scale down frequency based on utilization metrics [324]. However, DVFS is sub-optimal, imprecise, and has non-intuitive implementations in production [324] while overprovisioning GPU frequency leads to high power consumption. The optimal frequency is non-trivial to set for a GPU device that can serve different models or a mix of model partitions from different models [312] when considering stringent latency SLOs, throughput, and costs.

[C2] Second, serving non-deterministic generative models makes it hard to differentiate the negative impact of GPU frequency down-scaling from the inherent non-determinism in execution time. The non-determinism comes from the *autoregressive* nature of generative models. To process each request, one has to run multiple iterations; the output of each iteration is used as input in the following iteration. Worse, a first-come-first-serve (FCFS) request scheduling policy suffers from *head-of-line blocking* issues [20], leading to less power-saving headroom without SLO attainment violations.

[C3] Finally, unlike CPUs, GPUs do not support fine-grained (e.g., per-core) frequency scaling. Therefore, if the model or multiple model partitions (from different models) are placed onto a device, the frequency can only be reduced to the maximum frequency demand among all partitions. Power-unaware model partitioning and placement strategies [312] fail to unlock further power saving, even if the optimal frequency mentioned in [C1] can be figured out for each partition.

Our Work. This chapter addresses the above challenges by presenting μ-Serve, the first power-aware deep learning model serving framework with fine-grained model provisioning and GPU frequency scaling. The goal of μ-Serve is to maximize power efficiency while preserving the model-serving performance (i.e., SLO attainment). To achieve the goal, μ-Serve consists of the following main components:

- ***Power-aware fine-grained model provisioning.*** μ-Serve generates a model parallelism plan (i.e., model partitioning) for each model and then places model partitions onto devices in a *power-aware* manner. A model comprises a series of operators (e.g., multiply) over multidimensional tensors. Our key insight is that some operators are more *sensitive* [13] to frequency changes while others are not; those operators that are insensitive to frequency changes often contribute less to the end-to-end latency of serving a request. Based on this insight, we design a novel model provisioning algorithm (Section 6.3.1 and Section 6.3.2) by considering operator sensitivities to generate model partitions and ensure that each device contains partitions with comparable sensitivities. This algorithm addresses **[C3]** and maximizes power-saving opportunities.

- ***Speculative request serving.*** To address non-deterministic execution when serving generative models (i.e., **[C2]**), we introduce a lightweight *proxy model* that is fine-tuned to predict the execution times of each input query. This is based on our insight that a much smaller model can be quite good at understanding the output length of a large model. Based on the proxy model, we design a speculative shortest-job-first (SSJF) scheduler (Section 6.3.3) that improves the performance predictability of serving requests of autoregressive models. SSJF helps achieve higher SLO attainment and consequently more power-saving opportunities.

[13] We define sensitivity as the change in operator execution latency ($\Delta L > 0$) given the change in GPU frequency ($\Delta F < 0$), i.e., $-\Delta L/\Delta F$.

- ***SLO-preserving GPU frequency scaling.*** At runtime, μ-Serve uses a multiplicative-increase-additive-decrease (MIAD) algorithm (Section 6.3.4) to exploit the power-saving opportunities and dynamically scale GPU frequency while preserving performance SLOs, which addresses **[C1]**.

Offline Profiling and Online Orchestration. μ-Serve requires an offline phase before running online for power-aware model serving. In the offline phase, μ-Serve first profiles each *primitive operator* [325] to get its *sensitivity score*, indicating how operator execution latency changes with the change in GPU frequency. After obtaining a sensitivity score database, μ-Serve then generates power-aware model partitioning plans by utilizing model parallelism (Section 6.3.1) and deploys model partitions based on placement plans (Section 6.3.2) that create maximized power-saving opportunities. Such power-saving opportunities are then exploited at runtime by dynamic GPU frequency scaling based on workloads and SLO performance. In the online phase, μ-Serve serves model inference requests with a novel speculative shortest-job-first scheduler (Section 6.3.3) that addresses non-deterministic execution patterns of autoregressive models. To reduce power consumption while preserving model-serving performance SLOs, μ-Serve dynamically scales GPU frequency at each device with an MIAD algorithm (Section 6.3.4).

Results. We evaluate μ-Serve with a diverse set of deep learning models (including traditional non-Transformer models like CNNs, Transformers, and Transformer-based generative models) and production workloads on an 8-node 16-GPU cluster (Section 6.4). Evaluation results show that, compared to existing state-of-the-art serving systems, μ-Serve achieves a power reduction factor of 1.9–2.6× at varying rates, 1.5–2.3× at varying stringent SLOs, and 1.2–2× when scaling to higher numbers of devices, without SLO attainment violations.

Contributions. In summary, our main contributions are:

- A novel power-aware ML model serving framework that unlocks power-saving opportunities through fine-grained operator management and dynamic GPU frequency scaling.

- A speculative request scheduler with light proxy models that addresses non-determinism in generative models.

- An open-source implementation of μ-Serve and evaluation on real-world model datasets and production traces.

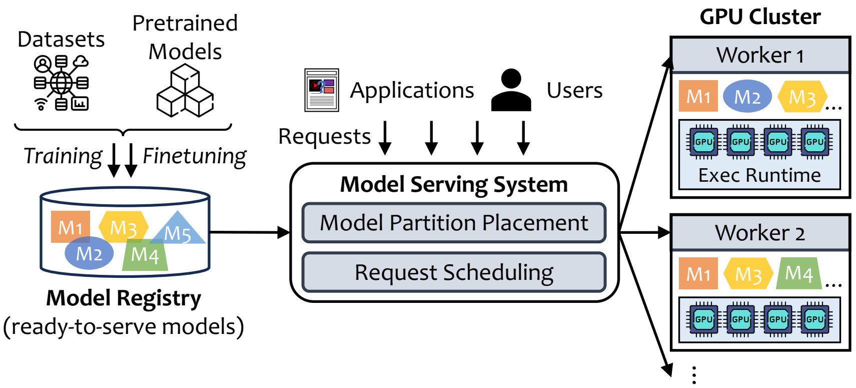


Figure 6.1: Model training and model serving.

6.2 BACKGROUND AND MOTIVATION

We begin with an overview of model-serving systems (Section 6.2.1), opportunities for power saving (Section 6.2.2), and non-deterministic execution of generative model workloads (Section 6.2.3).

6.2.1 Model Serving System Model

Model training is the process of building or learning a model from datasets, starting from scratch or pre-trained models [18]. The training process requires iterative *forward and backward passes*. After training, those ready-to-serve models are saved to a *model registry* (Fig. 6.1 left).

Model serving, or model inference, is to use the model to extract useful features from the inputs through a *forward pass* (Fig. 6.1 right). The structure of model-serving workloads follows a simple request-response paradigm where clients (either users or AI applications) submit requests for that model to a serving system, which schedules the requests, dispatches them to hardware devices (e.g., GPUs), and returns the results. Compared to model training, model serving does not involve complex backward computation and model weight updates but has more demanding performance requirements in inference latency (i.e., request completion time) and throughput. With the rise of pre-trained foundation models (e.g., in NLP and vision), it is expected that a model can be trained once and serve an extensive sequence of millions of inferences.

In this chapter, we focus on the Deep Neural Network (DNN) model (including CNNs [326], Transformers [327], and Transformer-based generative models [328]) serving on homogeneous GPU clusters (Fig. 6.1).

Model Parallelism and Partitioning. To satisfy user demand, model serving often must adhere to stringent SLOs on latency (e.g., 99% of the requests for a model must be finished within 500 ms). However, there can be significant and unpredictable burstiness in the arrival process of user requests [189, 312, 329]. Parallel model execution is necessary when attempting to satisfy the serving latency requirements or support large models that do not fit in the memory of a single device [312].

DNN models are composed of a series of *operators* (e.g., matrix multiplication and activation functions) over multidimensional *tensors*. There are two categories of model parallelism: *intra-operator* and *inter-operator* parallelism [330]. Intra-operator parallelism is to partition a single operator across multiple devices for parallel execution. It reduces the computation latency for serving a request but introduces the communication overhead of splitting the input and merging the output. Inter-operator parallelism is to partition a model's operator execution graph into multiple *stages* that can execute on multiple devices in a pipeline fashion. It allows the model to exceed the memory limitation of a single GPU device but it does not reduce the computation latency of a single request.

Power Efficiency. To the best of our knowledge, μ-Serve is the first power-aware DNN model-serving framework that aims to minimize power consumption while preserving the model-serving performance. State-of-the-art model-serving frameworks and commercial model-serving products use default GPU core frequency settings [324] for two main reasons. First, it is nontrivial to determine the optimal GPU frequency when serving diverse models or a mix of model partitions from different models when considering both stringent SLOs and costs. Second, GPU DVFS (dynamic frequency scaling based on utilization) is shown to be sub-optimal, imprecise, and has non-intuitive implementations in production [324].

Power optimization in ML model training [320, 321, 322, 323] has been active in the past few years. The central idea is to find the optimal GPU core frequency based on the valley trends when scaling frequencies (i.e., energy saving is maximized on the middle-level frequency). Building on top of that, EnvPipe [322] reduces frequencies at stages that are not on the critical path of the training pipeline with interleaving forward and backward passes. Perseus [323] identifies straggler stages (e.g., due to I/O bottleneck or hardware failures) and reduces frequencies to synchronize the speed of all other stages. However, these optimization tricks cannot be applied to model inference with a single forward pass. It gets challenging in model serving due to a mixture of model partition placement and dynamic request arrival patterns (unlike static and iterative training).

6.2.2 Opportunities for Power Saving

To understand if model serving also exhibits power-saving opportunities like model training, we start with an illustrative experiment to show how GPU frequency affects model serving latency. We use two NVIDIA Tesla V100 (16 GB) GPUs to serve each DNN model from Table 6.1. During the experiments, we reduce the GPU frequency from 1300 MHz to 200 MHz with a step size of 100 MHz. Fig. 6.2 (left) shows the end-to-end latency of serving a single request while Fig. 6.2 (right) shows the corresponding power efficiency.

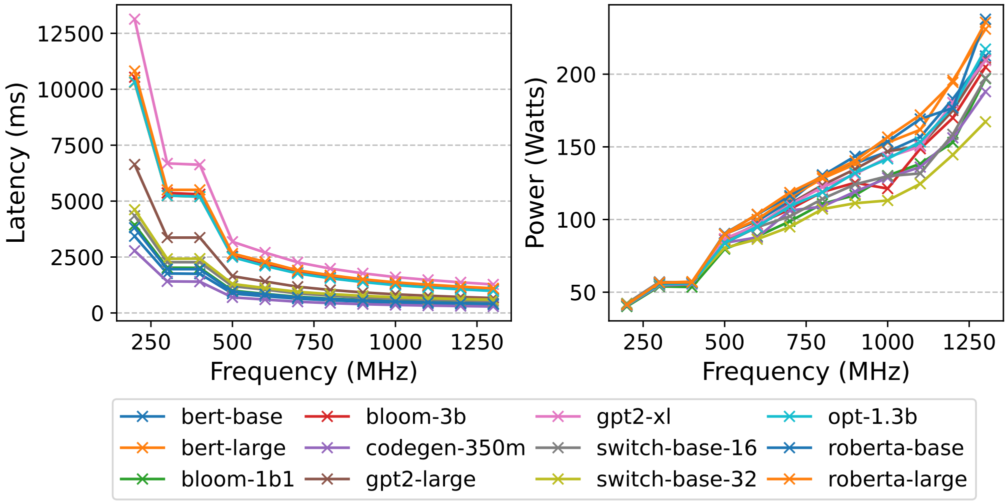


Figure 6.2: Model-serving latency and power consumption characterization at varying GPU core frequency levels.

The model-serving latency-frequency curves of all models exhibit a steep decline at low frequencies, but the decrease rate eventually approaches a horizontal asymptote at high frequencies. In contrast, the power curves for serving all models show a nearly linear relationship between the GPU frequency and the power consumption. This contrasting observation leads to a power-saving opportunity by reducing the GPU frequency when the SLO attainment is high. For example, serving `gpt2-large` can meet the SLO (1000 ms) even when the GPU frequency is reduced from 1300 MHz to 800 MHz. By doing so, the power consumption is reduced by a factor of 1.8× from 214 Watts to 120 Watts.

However, such power-saving opportunity varies as the SLO attainment depends on factors such as request arrival rate, burstiness, and SLO scales (as defined in [312]). In addition, coarse-grained GPU frequency scaling for an entire model cannot fully unlock the power-saving opportunity. We show in Section 6.3 how fine-grained model multiplexing and dynamic GPU frequency scaling can help based on the performance-power sensitivity and activation frequency of different operators.

6.2.3 Autoregressive Patterns

Traditional DNN models (e.g., ResNet and BERT) have deterministic execution patterns while Transformer-based generative models (e.g., GPT) are trained to generate the next token in an *autoregressive* manner. Therefore, to process a request to generative models, multiple *iterations* of the model have to be run; each iteration generates a single output token, which is then appended to the original input in the following iteration (except for the termination token `<EOS>`).

To illustrate this non-deterministic nature of serving model inference workloads due to autoregressive patterns, we analyze the output token length distribution in the LMSYS-Chat-1M dataset [331] which contains one million real-world conversations with 25 state-of-the-art LLMs. The output token length (N) dominates the execution time (T) of a request because $T = C + K * N$, where K is the latency to generate one token and C is the model-serving system's overhead including DNS lookups, proxies, queueing, and input tokenization. K depends on model optimization techniques (e.g., quantization) and execution environment (e.g., hardware), which are the same for all inputs. As shown in Fig. 6.3, the output token length of the model output varies significantly for the same model. The p95/p50 of the output token length for each model varies from 1.7 (`claude-1`) to 20.5 (`llama-13b`).

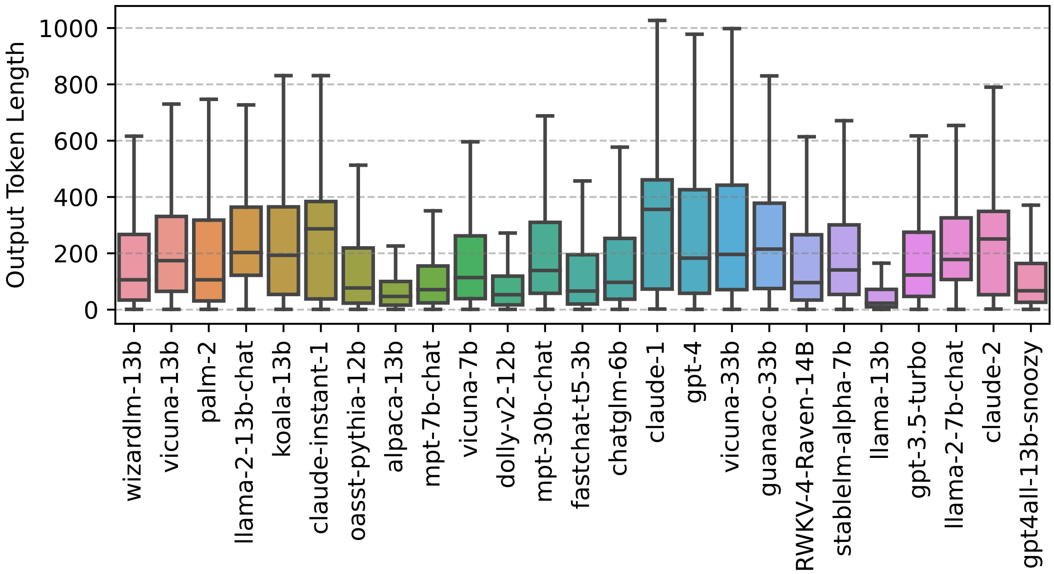


Figure 6.3: Output token length distributions of various large language models collected in the LMSYS-Chat-1M dataset.

Since model executions are non-deterministic, it is challenging to differentiate the negative impact of GPU frequency down-scaling from the inherent non-determinism. Worse, it suffers from *head-of-line blocking* issues when scheduling requests for model-serving as most state-of-the-art LLM inference systems use a first-come-first-serve (FCFS) scheduling policy. FastServe [317] uses a multi-level feedback queue to improve the average job completion time but it does not address the non-determinism nature.

6.3 μ-SERVE DESIGN AND IMPLEMENTATION

μ-Serve is a power-aware multi-model model-serving framework designed for a homogeneous GPU cluster that serves Deep Neural Network (DNN) models (including Transformers and Transformer-based generative models [328]) serving systems. From Section 6.2, we can see that there are several key challenges to improving power efficiency in DNN model serving:

- Determine the optimal SLO-preserving GPU frequency that minimizes power consumption (i.e., **[C1]**).
- Address non-deterministic execution patterns in generative models for performance predictability and avoiding head-of-line blocking (i.e., **[C2]**).
- Maximize the power-saving opportunity while GPUs do not support fine-grained frequency scaling (i.e., **[C3]**).

We propose μ-Serve to specifically tackle these challenges. The overall architecture of μ-Serve is shown in Fig. 6.4, which consists of an offline phase and an online phase. In the offline phase, μ-Serve first profiles each *primitive operator* [325] to get its *sensitivity score*, indicating how operator execution latency changes with the GPU frequency reduction. Based on the operator-level sensitivities, μ-Serve then generates power-aware model partitioning plans by utilizing model parallelism (Section 6.3.1) and deploys model partitions based on placement plans (Section 6.3.2) that maximize power-saving opportunities.

In the online phase, μ-Serve serves model inference requests while dynamically managing device frequency scaling. For each autoregressive model, μ-Serve uses a novel speculative shortest-job-first scheduler (as described in Section 6.3.3) that improves SLO attainment by avoiding head-of-line blocking and thus increases power-saving opportunities. For the remaining models, μ-Serve uses the default FCFS scheduling policy. To minimize power consumption while preserving model-serving performance SLOs, μ-Serve dynamically scales GPU frequency at each device with a multiplicative-increase-additive-decrease (MIAD) algorithm (as described in Section 6.3.4).

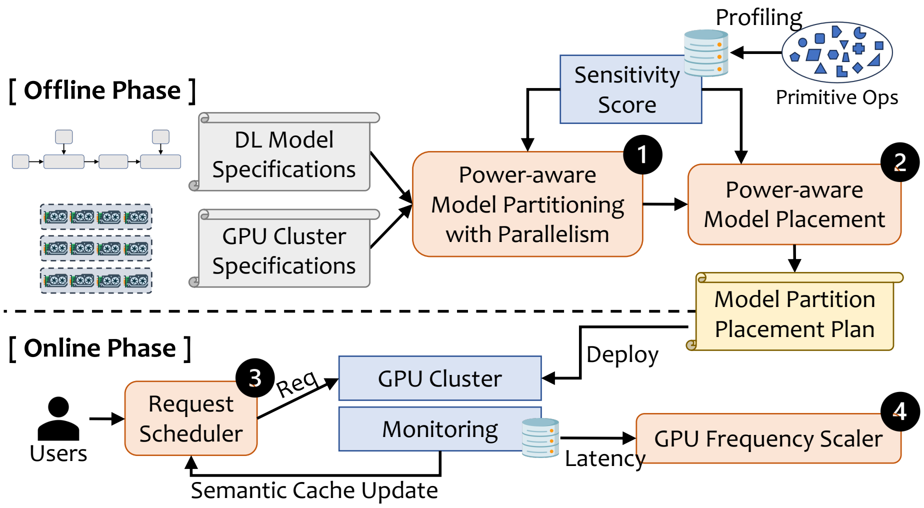


Figure 6.4: μ-Serve architecture overview and workflow.

6.3.1 Power-aware Model Parallelism

Model parallelism is necessary to enable running larger models, or larger batches of inputs, and to serve bursty workloads with better latency and throughput [312, 330]. μ-Serve parallelism module (1 in Fig. 6.4) partitions each ML model (by using model parallelism) to generate model partitions that can *maximize power efficiency* while considering the latency and throughput trade-offs so that the placement module (2 in Fig. 6.4) can choose the best combination for all models in the whole cluster. To achieve this, μ-Serve relies on profiled *operator sensitivity scores* and AlpaServe [312], an auto ML model parallelization system for inference.

As described in Section 6.2.1, when compiling and deploying DNN models for request-serving, each model is commonly defined as a computational (dataflow) graph where nodes are *operators* and edges are *tensors* (or data) [332]. Example operators include matrix multiplication and `relu` or `tanh` activation functions). Serving a request means the completion of the dataflow in the execution graph. Our key insight is that (1) some operators are more *sensitive* to GPU frequency changes (i.e., *sensitive operators*) while others are not (i.e., *insensitive operators*); and (2) insensitive ones often contribute less to the end-to-end latency of serving a request. Since the minimum GPU frequency that a model partition can tolerate is determined by the most sensitive operators, μ-Serve avoids mixing sensitive and insensitive operators into the same partition.

Operator Sensitivity Score Profiling. Since the model graphs are represented in XLA's HLO format [325], a backend-agnostic intermediate representation (IR), we choose to profile only the *primitive operators* [14] defined in HLO that are shared by all models. We define the sensitive score of each operator as $\Delta Latency/\Delta Frequency$ where $\Delta Latency$ is the increase in execution time (normalized) of the operator when decreasing the GPU core frequency from the default setting to half of the default frequency and $\Delta Frequency$ is the frequency decrease. We adopt the tensor dimensions from AI-Matrix [333], a production dataset released by Alibaba. When profiling primitive operators, we run each operator independently with each tensor dimension repeated 5000 times and take the average of all runs. Profiling of primitive operators happens offline and the obtained sensitivity score database is then used in model parallelism and placement plan generation.

Model Parallelism Plan Generation. Since different parallelization configurations (intra-/inter-operator parallelism) have different latency and throughput trade-offs, μ-Serve extends AlpaServe [312] to run an auto-parallelization compiler to generate a list of possible configurations for every single model and let the placement module (Section 6.3.2) choose the best combination for all models in the whole cluster.

At the core of AlpaServe runs an optimization algorithm with one inter-operator pass (based on dynamic programming) and one intra-operator pass (based on integer linear programming) to generate efficient model parallel partitions. AlpaServe treats each operator as the smallest unit for partitioning. To prevent operators with similar sensitivities from being split we do not change the AlpaServe auto-parallelism algorithm but extend AlpaServe with a new abstraction, i.e., *operator cluster*, which is a cluster of operators that share similar sensitivity scores. μ-Serve treats each operator cluster instead of each operator as the partitioning unit. As shown in Alg. 6.1, generating operator clusters can be achieved by taking the model specifications and profiled sensitivity scores and grouping neighbor operators with similar sensitivity scores (below a threshold) together. The output is the model parallelism plan (i.e., partitions and their memory demands for a given model) that optimizes the per-request execution latency. We leave the decision of model placement (which also happens offline) to Section 6.3.2.

6.3.2 Power-aware Model Placement

Given a list of model partitions (generated by Section 6.3.1) and a fixed GPU cluster, model placement is to map each model partition to a GPU device within the memory constraint. The goal of model placement is to maximize SLO attainment (i.e., the percentage

[14] As a unifying abstraction for multiple frameworks (e.g., Pytorch and TensorFlow) and hardware platforms (CPUs, GPUs, and TPUs), XLA summarizes common DL operators into around 100 primitive operators.

Algorithm 6.1 Operator Clustering

```
Require: Model specification $M$, Operator sensitivity score database $D$ (a map of <operator, score> pairs)
procedure OperatorClustering($M$, $D$)
    $clusters \leftarrow \varnothing$
    $currCluster \leftarrow \varnothing$
    $currCluster$.add($M$.head)
    $clusterScore \leftarrow$ avg($currCluster$)
    for each $op$ in $M.operators$ do
        if abs($D$.get($op$) - $clusterScore < THRES$) then
            $currCluster$.append($op$)
            $clusterScore \leftarrow$ avg($currCluster$)
        else
            $clusters$.append($currCluster$)
            $currCluster \leftarrow$ newList([$op$])
        end if
    end for
    Return $clusters$
end procedure
```

of requests served within SLO). Power-aware model placement module (❷ in Fig. 6.4) takes model partitions and the GPU cluster specification as inputs and generates a model partition placement plan that preserves SLOs while maximizing power-saving opportunities.

However, model placement is typically modeled as a bin-packing problem and is a computationally NP-hard problem [334]. For scalability and serving potentially a larger device cluster, we partition the cluster into several groups of devices (same as in AlpaServe). Each group of devices selects a subset of models to serve. Finding the optimal ⟨model partition, device group, device⟩ is a combinatorial optimization problem with a configuration space growing exponentially with the number of devices and the number of models. μ-Serve first selects the ⟨model partition, device group⟩ configurations that maximize SLO attainment based on AlpaServe [312] and then generates ⟨model partition, device⟩ configurations that maximize the power-saving opportunities. After deploying the placement plan, the offline stage is completed.

Mapping Models to Device Groups. μ-Serve runs the model placement algorithm in AlpaServe [312] that (1) uses a simulator-guided greedy algorithm to decide which models to select for each group, and (2) enumerates group partitions and model-parallel configurations to pick the best ⟨model, device group⟩ placement that maximizes SLO attainment.

Mapping Model Partitions to Devices. For each device group, given a list of devices and model partitions that are assigned to the device group from the last step, μ-Serve

Algorithm 6.2 Model Partition Placement

```
Require: Device group $DP$ (list of GPUs), Model partition plan $MP$ (multi-model), $D$ (same in Alg. 6.1), $CT$ (maximum number of devices to place to))
procedure ModelPlacement($DP$, $MP$, $D$)
    $bins \leftarrow \varnothing$                          ▷ a map of device to a list of partitions
    $modelBinsCount \leftarrow \varnothing$
    for each $model$ in $MP$ do
        // Sort partitions by sensitivity scores
        for each $p$ in $model$.getSortedPartitions() do
            $bestDevice$ = None
            $bestScore = INFINITY$
            $selectedBins$ = getAvailableBins($bins$, $p$)
            for each $d$, $partitions$ in $selectedBins$ do
                if $modelbinsCount[model] < CT$ then
                    // Get similarity in sensitivity scores
                    $score \leftarrow$ getSimilarity($p$, $partitions$)
                    if $bestScore < score$ then
                        $bestScore = score$
                        $bestDevice = d$
                    end if
                end if
            end for
            $bins[bestDevice]$.append($p$)
            $modelBinsCount[model]$ += 1
        end for
    end for
    Return $bins$
end procedure
```

generates ⟨model partition, device⟩ placement plans that maximize the power-saving opportunities. Based on our insight that (1) operators have diverse sensitivity to GPU frequency scaling and (2) insensitive operators contribute less to the end-to-end latency, the intuition behind μ-Serve's model placement algorithm is to group operators with similar sensitivities together. Since how much the GPU frequency can be reduced is determined by the *most sensitive* model partition on that device, μ-Serve avoids co-locating a mix of sensitive and insensitive partitions onto the same GPU devices.

The overall procedure is shown in Alg. 6.2. μ-Serve first estimates the sensitivity scores of a model partition based on the weighted sum of operator sensitivity scores (using the operator count and tensor dimensions as the weights). All model partitions are then sorted based on the estimated sensitivity scores. For each model partition in the sorted list, we get the feasible "bins" or devices (by checking memory availability) and assign the partition to

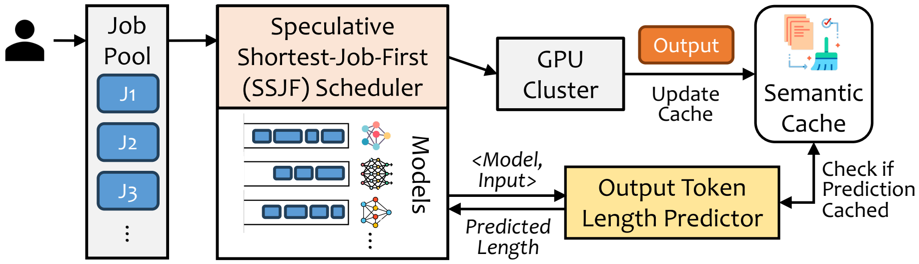

Job Pool
J1
J2
J3
Speculative Shortest-Job-First (SSJF) Scheduler
Models
GPU Cluster
Output
Update Cache
Semantic Cache
<Model, Input>
Predicted Length
Output Token Length Predictor
Check if Prediction Cached

Figure 6.5: Workflow of the speculative scheduling in μ-Serve.

a bin that has the best *fitting score*. The fitting score for ⟨partition p, bin b⟩ is defined as the similarity between the sensitivity score of p and the weighted average of the sensitivity scores of all existing partitions in b. To avoid spreading partitions from the same model to too many devices (and thus inducing high communication overhead), we enforce an upper limit CT on how many bins a model can be packed into. CT is set based on the empirical results of the maximum number of devices to partition onto without SLO violations. The outcome is the model placement plan with a list of bins and corresponding model partition assignment. μ-Serve takes the model placement plan for deployment, which completes the offline stage.

6.3.3 Scheduling

User requests are dispatched to corresponding models at runtime. For each model instance, a request queue is maintained, and μ-Serve uses a *speculative shortest-job-first (SJF) scheduler*(❸ in Fig. 6.4) to decide the request execution order of autoregressive models and a default FCFS scheduler for traditional deep learning models. SJF alleviates the non-determinism of generative models and the head-of-line blocking issue in FCFS scheduling. We must have knowledge or a good estimate to deploy SJF. As shown in Fig. 6.5, μ-Serve's speculative SJF scheduler relies on the prediction from an *output token length predictor*. The prediction is used to estimate the job execution time in SJF as the output token length dominates the execution time (linear relation) as described in Section 6.2.3. The predictor relies on a lightweight proxy model based on our insight that a small model is already quite good at predicting the lengthiness of a significantly larger (>20×) model's output. In addition, we use a *semantic cache* to improve the prediction accuracy of hot input queries.

Output Token Length Predictor. Given the user input to a certain generative model, our predictor tries to predict the length of the model's response in terms of the number of tokens because the model inference time is linearly related to the output token length (Section 6.2.3). We use BERT as a proxy model and build a multi-class classifier based on BERT as the output token length predictor. Specifically, we take the last layer hidden state of the first token (i.e., `CLS`) from the BERT output, and pass it through an additional two-layer feed-forward neural network to generate the prediction (Fig. 6.6). As suggested by the original BERT paper [327], the final hidden state corresponding to this token is used as the aggregate sequence representation for classification tasks. We could get better classification performance by averaging or pooling the sequence of hidden states for the whole input sequence. However, taking the whole sequence is more compute-intensive and therefore not worth the marginal gains.

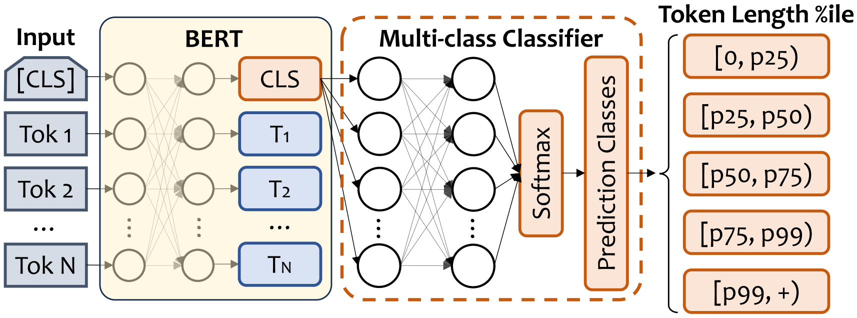


Figure 6.6: Output token length predictor in μ-Serve.

Predictor Training. Our predictor adopts the pre-trained BERT model weights as provided by [327] and we further fine-tune the weights (together with the additional two-layer classifier network) on the LMSYS-Chat-1M training data [331]. Pre-training on a large corpus of unlabeled text gives BERT some basic understanding of human languages, while the fine-tuning step allows BERT to learn the *task-specific* knowledge (in our case, the output token length of a generative model given the input query). Our fine-tuning has two phases: In the first phase, the parameters of both BERT and the classifier are tuned, while in the second phase, we fix the weights of BERT and only update the classifier's parameters. Such a two-phase regime turns out to achieve a good balance between prediction accuracy and training efficiency.

Since predicting the exact length of the output can be difficult, we treat the token length prediction problem as a coarse-grained (multi-class) ordinal classification task [335]. Specifically, we characterize the output token length percentiles of each generative model based on its training samples and categorize the length labels of each model into 5 classes, namely [0, p25), [p25, p50), [p50, p75), [p75, p99), and [p99, +). We convert the classes into 5 integers $\{0, 1, 2, 3, 4\}$ and let our predictor output a real value as the prediction. We apply a mean squared error (MSE) training loss to minimize the distance between the real-valued predictions and the integer-valued class labels. During inference, we round the prediction value to the nearest integer in $\{0, \ldots, 4\}$. As shown in Sec. 6.4.3, such an ordinal classification approach outperforms pure regression- or classification-based solutions.

To handle the requests sent to diverse models with diverse output verbosity (as shown in Fig. 6.3), we train a single predictor shared by all the models but additionally convert the model names into one-hot feature vectors and include them as part of the input to our predictor to let it distinguish between different models. A shared predictor leads to some degradation in the prediction accuracy when compared to having a separate predictor for each model. Nevertheless, using a single predictor largely reduces the maintenance cost and improves the training sample efficiency by allowing the predictor to simultaneously learn from the samples of multiple models.

Choices of Classification. The granularity of classification (i.e., how many classes to predict) is determined by the end-to-end scheduling results (in latency and throughput) instead of the prediction accuracy. It is because of our observation that better prediction accuracy does ***not*** necessarily lead to better scheduling performance. Empirical evaluations on our datasets suggest that as the number of classes rises, prediction accuracy drops (because it gets harder to predict more fine-grained output lengths), but scheduling performance first rises and then falls, peaking when there are 5 classes. We attribute this observation to the fact that the scheduler benefits from more fine-grained prediction but receives negative impacts when prediction is worse, i.e., when predicting $>$5 classes.

Use of Semantic Cache. The μ-Serve scheduler includes a simple *semantic cache* based on GPTCache [336] to store the ground truth output token sequence length of hot inputs. Instead of going through the predictor to get an estimated output length prediction, μ-Serve retrieves the cached length for the input that triggers a cache hit. Caching has been commonly used to reduce frequent and computationally expensive data accesses. In μ-Serve, a semantic cache is a $\langle$key, value$\rangle$ memory buffer where keys are *embeddings* and values are output lengths. Embeddings are generated by embedding models that map text inputs into a low-dimensional continuous vector space and are stored in a vector database [336]. For each input, the most similar cached input is retrieved using a similarity evaluation function to determine if the cached input matches the input query semantically. The cached ground truth length is then used by the scheduler. If there is a cache miss, the scheduler still uses the predicted output length. See Section 6.3.5 for implementation details.

6.3.4 GPU Frequency Scaling

μ-Serve's GPU frequency scaler (❹ in Fig. 6.4) consists of a monitoring and a frequency scaling component. The former monitors per-model request-serving latency while the latter determines appropriate GPU frequency scaling actions for each GPU device. We adopt a

Algorithm 6.3 GPU Frequency Scaling

```
Require: GPU Device $GD$, Model Partitions $MP$ (a map of ⟨model, partition⟩ pairs placed on $GD$)
procedure GPUFrequencyScaling($GD$, $MP$)
    // Start from the default high frequency
    $freq = DEFAULT$
    $GD.setFrequency(freq)$
    while $True$ do
        $action = SAME$                                ▷ one of $\{UP, DOWN, SAME\}$
        for each $p$ in $MP$ do
            $target = p$.getModelSLO()
            $actual = p$.getModelLatency()
            $slack = (target - actual)/target$
            if $slack < 0.05$ then
                $action = UP$                          ▷ requires up-scaling
                break
            end if
            $latency = p$.getWeight() * $actual$
            $degradation = latency * STEP/freq$
            if $degradation/target < slack - 0.05$ then
                $action = DOWN$                        ▷ down-scaling
            end if
        end for
        if $action == UP$ then
            $freq = min(MAX, freq * 2)$
        else if $action == DOWN$ then
            $freq = max(MIN, freq - STEP)$
        end if
        $GD.setFrequency(freq)$
    end while
end procedure
```

multiplicative-increase-additive-decrease (MIAD) algorithm [337] for GPU scaling based on the feedback signal of the slack between the target SLO latency and the actual latency, inspired by the AIMD algorithm in congestion control. The intuition behind adopting MIAD is that we want to be conservative when scaling down GPU frequencies to avoid interruptions to SLO attainment (addictive decrease) while being aggressive when scaling up GPU frequencies to respond fast to workload spikes.

As shown in Alg. 6.3, the scaler starts from the default frequency (usually the maximum frequency). After initialization, model latencies are sampled every 500 ms, and the frequency scaling step is determined at every iteration by scanning through all model partitions' latency *slack* (as defined in L10 in Alg. 6.3). For each model partition, any negative or small slack leads to an up-scaling action (we set the threshold = 0.05). Otherwise, we estimate the degradation of down-scaling the frequency based on the linear relationship between frequency and execution latency. A larger-than-degradation slack results in a down-scaling action. Ultimately, after scanning through all model partitions, an up-scaling action doubles the current frequency while a down-scaling action decreases the current frequency by $STEP$. Note that when the current frequency reaches the maximum, but scaling-up is repeatedly called, it can be treated as a signal to replicate the model instance. Each iteration is performed every second.

6.3.5 Implementation

μ-Serve is implemented with 6.1K lines of code in Python on top of an existing model-parallel inference system, AlpaServe [312]. We extend its auto-parallelization algorithms to get the power-aware model-parallel strategies and placement plans. In μ-Serve's scheduler, the semantic cache is implemented with GPTCache [336]. We use the HuggingFace embedding function, the `NumpyNormEvaluation` similarity evaluation function (with a default similarity threshold of 0.8), the SQLiteCache cache manager, and the FAISS vector database. To avoid starvation in scheduling, μ-Serve adopts aging [338] to promote jobs with long waiting times. Preemption could help correct previous suboptimal decisions with a least-slack-time-first policy. However, μ-Serve does not adopt preemption due to its added complexity in context switch [339], memory management [318], and cache replacement policies [317] (for key-value cache in serving LLMs).

6.4 EVALUATION

Our experiments addressed the following research questions:

§6.4.2 Does μ-Serve achieve power-saving (and how much) while preserving SLO attainment?

§6.4.3 How much does the proxy-model-based scheduler in μ-Serve improve latency and throughput?

§6.4.4 How robust is μ-Serve's power-aware model partitioning and placement under model variations?

Table 6.1: Details of the models used in experiments.

Model	# of Params	Size	Latency	AR?
ResNet-50	25M	0.2 GB	51 ms	No
BERT-base	110 M	0.5 GB	123 ms	No
BERT-large	340 M	1.4 GB	365 ms	No
RoBERTa-base	125 M	0.5 GB	135 ms	No
RoBERTa-large	355 M	1.4 GB	382 ms	No
OPT-1.3b	1.3 B	5.0 GB	1243 ms	Yes
OPT-2.7b	2.7 B	10.4 GB	2351 ms	Yes
GPT2-large	774 M	3.3 GB	832 ms	Yes
GPT2-xl	1.5 B	6.4 GB	1602 ms	Yes
CodeGen-350m	350 M	1.3 GB	357 ms	Yes
CodeGen-2b	2.0 B	8.0 GB	2507 ms	Yes
Bloom-1b1	1.1 B	4.0 GB	523 ms	Yes
Bloom-3b	3.0 B	11.0 GB	1293 ms	Yes
Switch-base-16	920 M	2.4 GB	348 ms	Yes
Switch-base-32	1.8 B	4.8 GB	402 ms	Yes

6.4.1 Experiment Setup

Models. Since μ-Serve is a general ML model-serving framework, we consider traditional non-Transformer models, Transformers, and Transformer-based generative models for evaluation. For each model family, we select several most commonly used model sizes and variants (to mimic different fine-tuned versions) for experimentation. Table 6.1 provides details about model sizes and inference latency on testbed GPUs. The latency is measured for a single query with a sequence length of 512 on a single GPU. AR stands for autoregressive.

Workloads. We use the Microsoft Azure function traces [340] and production traces from a private datacenter dedicated to ML workloads in SenseTime [341] to drive the inference workloads. Since the SenseTime dataset contains both ML training and inference workloads (without distinction), we select the traces with job completion time < 5 seconds and the number of used GPUs ≥ 1 to ignore training jobs and admin query workloads (CPU workloads, e.g., training progress check). For each model in Table 6.1, we round-robin serverless functions and ML inference jobs to generate traffic. For vision model inputs, we use the ImageNet dataset [342]. For language model inputs, we use the LMSYS-Chat-1M dataset [331] which contains one million real-world conversations.

Metrics. We use SLO attainment (i.e., the percentage of requests served within the latency SLO) and power consumption (in Watts) as the major evaluation metrics since the goal of μ-Serve is to minimize power consumption while preserving SLO performance. We adopt the definition of *SLO Scales* from AlpaServe [312] which refers to different multiplies of the SLO latency of the model request. The SLO latency of the model request is measured

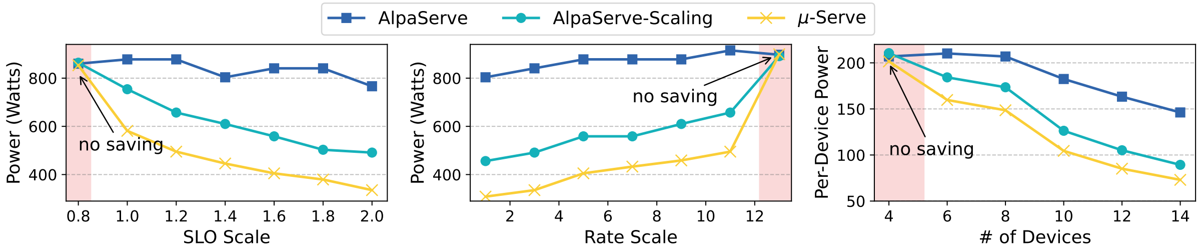


Figure 6.7: Power saving evaluation.

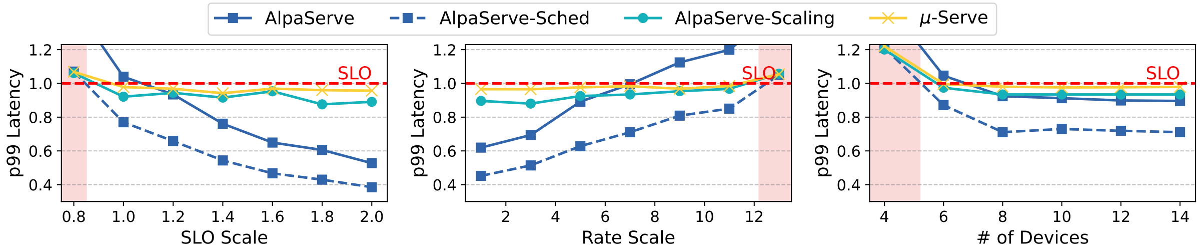


Figure 6.8: SLO preservation evaluation.

as the single-request latency with the parallelism plan generated by AlpaServe. We are also interested in model-serving throughput and how these measures change with varying numbers of devices, request arrival rates, and traffic burstiness.

Testbed. We deploy μ-Serve on a cluster with 8 nodes and 16 GPUs. Each node is an IBM Cloud `gx2-16x128x2v100` instance with 2 NVIDIA Tesla V100 (16GB) GPUs. Each GPU supports a maximum Streaming Multiprocessor (SM) frequency of 1380 MHz and a minimum of 200 MHz.

6.4.2 End-to-end Results

In this section, we compare μ-Serve against three baselines (AlpaServe [312], AlpaServe-Sched, and AlpaServe-Scaling) when serving widely used open-source large deep learning models on publicly available workload traces (as described in Section 6.4.1). AlpaServe is a state-of-the-art model-serving system that generates model parallelism and placement plans to maximize SLO attainment under varying workload conditions. AlpaServe-Sched is AlpaServe with the FCFS scheduler replaced with μ-Serve's scheduler (Section 6.3.3) but without GPU frequency scaling. AlpaServe-Scaling deploys model partitions according to parallelism and placement plans from AlpaServe and runs μ-Serve's GPU frequency scaling algorithm (Section 6.3.4) and scheduler at runtime. We are interested in the power-saving comparison and knowing whether μ-Serve achieves power-saving while preserving the SLO attainment.

We deploy all models (in Table 6.1) in the experiments. Each model instance is driven by an independent request generator using a randomly selected trace set as described in Section 6.4.1. The memory consumption of all models (including activation and runtime contexts) is 62.5 GB which can be fit onto 4 devices.

Power Saving. Fig. 6.7 shows the power consumption of serving all models with varying SLO scales, (combined) arrival rates, and the number of devices. Note that when SLO attainment is violated, there is no power saving (as shown in the red-shaded area, e.g., an SLO scale of 0.8) because the device frequency is the same for all approaches. When SLO attainment is met, μ-Serve reduces the power consumption in AlpaServe by 1.51–2.29× at different SLO scales. At different arrival rates, the reductions in power consumption are 1.85–2.61×, compared to AlpaServe. The larger the SLO scale (less stringent SLO) and the smaller the arrival rate, the more the power saving. We find that adding dynamic GPU frequency scaling on top of AlpaServe has already been able to reduce power consumption by a factor of 1.1–1.5× and 1.4–1.7× at varying SLO scales and rates. μ-Serve further improves the power saving by an average of 1.4× compared to AlpaServe-Scaling. We find that the improvement does not vary with SLO scales or arrival rates because it is the statically set model partition placement strategies that cause different power savings from the same GPU frequency scaling approach.

In the experiments with varying numbers of devices, we replicate each model instance (so now the total model memory demand is up to 8 devices) and increase the number of devices from 4 to 14. Fig. 6.7 (right) shows the per-device average power consumption. When there are no idle devices (i.e., at 6 and 8 devices), the power saving from μ-Serve compared with AlpaServe is 1.23–1.39×. When there are more idle devices, all per-device power consumption curves start to decrease. Since μ-Serve and AlpaServe-Scaling actively down-scale the GPU frequency, the power saving increases to 1.75–2× and 1.44–1.64× compared to AlpaServe.

AlpaServe-Sched is not shown in Fig. 6.7 since we observe that it has the same power consumption as AlpaServe. This is because only the execution ordering of the requests is different between the two baselines and there is no idle time. Therefore, the scheduler does not have an effect on power saving.

Model Serving Performance. We evaluate the SLO attainment to understand what the cost of μ-Serve is while achieving power savings. We measure the 99th-percentile (p99) latency of serving a request for each model and normalized it to the SLO latency. When the normalized p99 latency is less than 1, it means the SLO attainment is greater than 99%. On the other hand, the model serving system does not meet the SLO attainment of 99%. Fig. 6.8 shows the normalized p99 latency of serving all models with varying SLO scales, (combined) arrival rates, and the number of devices.

Same as in the power-saving evaluation, the red-shaded area indicates that the 99% SLO

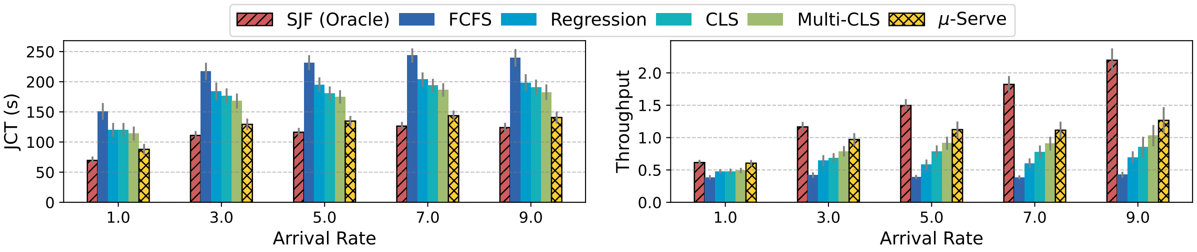


Figure 6.9: Scheduler evaluation under varying arrival rates.

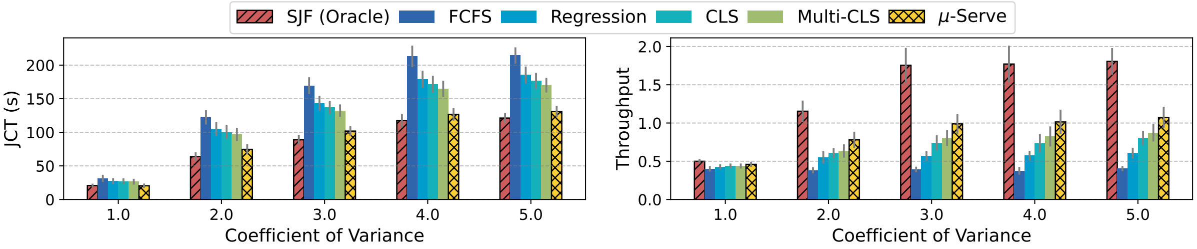


Figure 6.10: Scheduler evaluation under varying request burstiness.

attainment of AlpaServe-Sched is not met. Therefore, the three approaches AlpaServe-Sched, AlpaServe-Scaling, and μ-Serve converge to nearly the same p99 latency because the device frequencies are fixed at the maximum. When there is no SLO violation in AlpaServe-Sched, μ-Serve and AlpaServe-Scaling both meet the 99% SLO attainment. μ-Serve has 2–9% higher p99 latency than AlpaServe-Scaling. By replacing FCFS with μ-Serve's scheduler, AlpaServe-Sched achieves 26–31% lower p99 latency. In the experiments with varying numbers of devices (as shown in Fig. 6.8 (right)), since serving all models requires at least 8 devices, there are SLO attainment violations at 4 and 6 devices. At 6 devices, only AlpaServe fails to meet the SLO attainment. At 8 devices, all approaches meet the SLO attainment and μ-Serve has 5% higher p99 latency compared to AlpaServe-Scaling, which also holds when there are idle devices (i.e., at 10–14 devices).

6.4.3 Scheduling

In this section, we evaluate the output length predictor and scheduler performance. Additional ablation studies on the scheduler can be found in Appendix B.

Token Length Predictor Evaluation. We evaluate our output token length predictor on the LMSYS-Chat-1M dataset. Overall, our predictor achieves an average accuracy of 0.42 across 25 models (compared to the accuracy of 0.2 for a random guess). As shown in Fig. 6.11, despite using a shared predictor, our predictor achieves rather balanced performances across different models, with the prediction accuracy ranging from 0.33 for `llama-13b` to 0.50 for `wizardlm-13b`. Training a customized predictor for each individual model leads to slightly

Table 6.2: Output token length predictor evaluation results.

Metrics	Ord. CLS (MSE)	Ord. CLS (L1)	CLS	REG (MSE)	REG (L1)
Accuracy ↑	0.4290	0.4272	**0.4599**	0.4014	0.4383
F1 Score ↑	0.3563	0.3332	**0.4021**	0.3318	0.3477
L1 Error ↓	0.7587	0.7480	N/A	0.6077	**0.5557**
MSE ↓	0.9859	1.0857	N/A	**0.8077**	0.8996

better accuracy. For instance, a customized predictor for `vicuna-13b` achieves an accuracy of 0.429 (>0.42 for a general predictor). However, training customized predictors can incur more training and model management overhead. Therefore, we do not proceed with customized predictors.

We also conduct ablation studies to investigate the effectiveness of our ordinal-classification-based token length predictor. In Table 6.2, we compare our predictor with multiple alternative approaches, including ordinal classification with L1 loss, (standard) classification with cross-entropy loss and resampling, and regression with MSE & L1 loss. Interestingly, we find that higher prediction accuracy ***does not*** necessarily lead to better scheduling performance (e.g., lower latency or higher throughput). Instead, it matters more to the scheduler that the predictor can rank any two inputs correctly. For instance, a predictor with higher mean accuracy may mispredict several long outputs to be very short, leading to HoL and much worse latency. Table 6.2 shows that our ordinal classifier with MSE loss fails to outperform the alternative predictors in any metrics. However, as we will show in the scheduler evaluation next, μ-Serve's scheduler benefits more from our chosen predictor despite a slightly lower accuracy of 0.4290 (< 0.4599 compared to CLS).

Scheduler Evaluation. We evaluate the scheduling performance regarding the job completion time (JCT) and the serving throughput to understand if prediction accuracy is sufficient. Our comparison baselines include (1) first-come-first-serve (FCFS), which is the default scheduler in state-of-the-art model-serving frameworks such as AlpaServe [312] and Orca [316], and (2) shortest-job-first (SJF) with oracle output token prediction. Fig. 6.9 and Fig. 6.10 show the scheduling performances with varying request arrival rates and burstiness. We also compare with three alternative designs, i.e., SJF with regression predictor, binary-class predictor, and multi-class predictor (with no semantic cache). With varying request rates, μ-Serve reduces JCT by 41.3% compared to FCFS and increases the throughput by 2.5×. In comparison, the oracle SJF reduces the JCT by 49.8% and increases the throughput by 3.6×. Under different burstinesses, μ-Serve reduces JCT by 38.6% compared to FCFS and increases the throughput by 2.2×. In comparison, the oracle SJF reduces the JCT by 43.8% and increases the throughput by 3.6×. With semantic cache, μ-Serve improves JCT

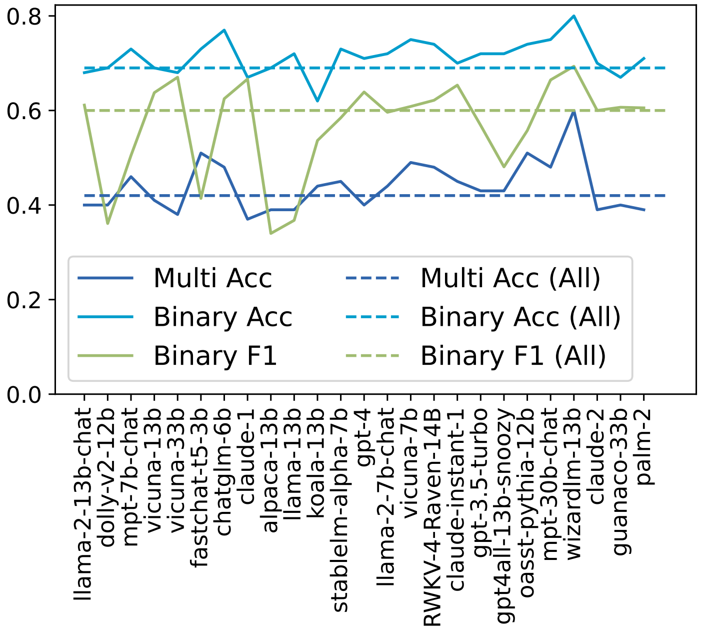


Figure 6.11: Predictor accuracy across all models.

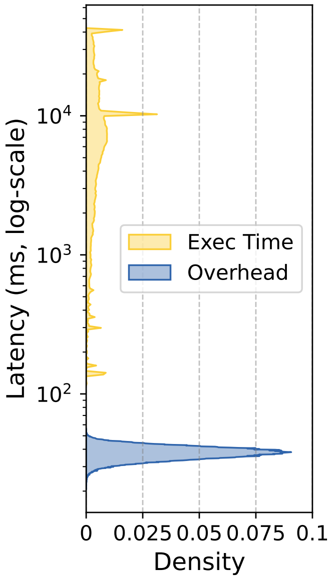


Figure 6.12: Overhead analysis of the proxy model.

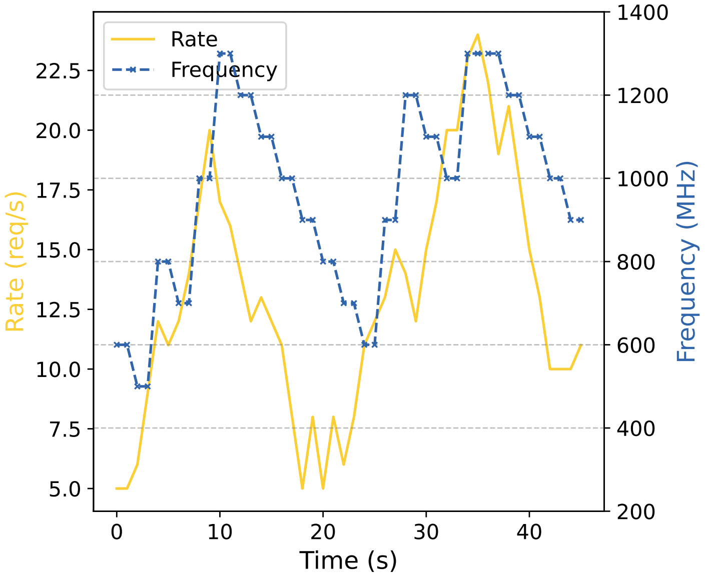
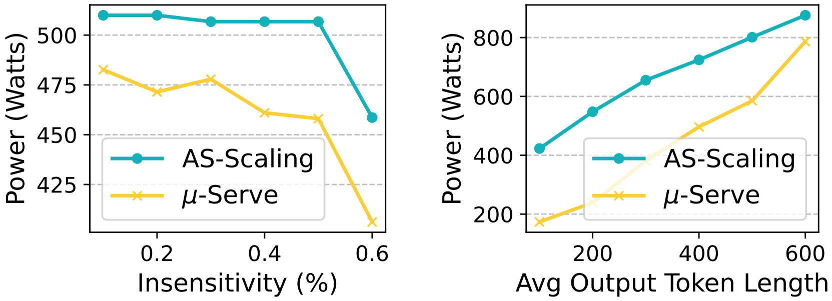


Figure 6.13: Frequency scaling.

by 23% and increases throughput by 1.2× (i.e., the comparison with Multi-CLS).

Predictor Overhead. We evaluate the latency overhead of the predictor for autoregressive models (as indicated in Table 6.1) because the output token prediction is only on the critical path of serving autoregressive models. The latency overhead includes both semantic



(a) Insensitivity (%).

(b) Output variation.

Figure 6.14: Robustness analysis.

cache lookup and proxy model prediction. Fig. 6.12 compares the predictor latency overhead and model request execution time. Note that the Y-axis is in the log scale. The average model execution time is 9.8 s, and the p5th execution time is 360 ms. The average predictor latency is 37.6 ms (0.4% of the total latency), and the maximum is 56.3 ms, which is less than the minimum model execution time of 120 ms. Therefore, we conclude that the predictor introduces negligible overhead to the end-to-end model-serving latency of autoregressive models, which supports the design and benefits of using a lightweight proxy model in μ-Serve.

Support for Request Batching. Adaptive batching [310, 313] has been used to maximize throughput while meeting latency objectives [313]. μ-Serve supports various batching techniques. More evaluation results can be found in Appendix B.3

6.4.4 Model Partitioning and Placement

In this section, we aim to understand the power-saving opportunity concerning the percentage of insensitive operators in a model, output non-determinism, and workload variation.

Power Saving Opportunity vs. Insensitivity Ratio. Power saving stems from the key insight that operators have diverse sensitivity to GPU frequency scaling, and thus, insensitive operators are leveraged to down-scale GPU frequency while preserving the SLO attainment. We define insensitive operators to be those model operators with sensitivity scores less than 1. Fig. 6.14 (a) illustrates how power-saving opportunities vary with the percentage of insensitive operators in different open-source models (up to 60%). Compared with AlpaServe-Scaling, the power reduction increases by 2.2× when the percentage increases from 10% to 60%.

Robustness to Model Execution Non-determinism. Fig. 6.14 (b) shows the relationship between power saving and the model output length. We find that as the average model output length increases (i.e., more iterations per model request), the power reduction of μ-Serve compared to AlpaServe decreases from 59% to 10% because of more intensive model executions (and thus fewer power-saving opportunities).

Robustness to Workload Variations. As shown in Section 6.4.2, μ-Serve achieves power saving while preserving SLO attainment by dynamic GPU frequency scaling. In reaction to workload variations, Fig. 6.13 illustrates the scaling actions of the MIAD algorithm in μ-Serve with the changes in request arrival rates. μ-Serve adopts an aggressive (multiplicative) frequency increase policy and conservatively decreases the frequency at each step (e.g., during the time 10–20 seconds). Note that if the frequency stays at the maximum frequency but there are continuous SLO attainment violations, a scale-out signal can be sent to the model autoscaler (which μ-Serve does not implement), left to the discussion in Section 6.6.

6.5 RELATED WORK

General Model-Serving Systems. The emergence of ML applications motivates the prevalence of ML model-serving systems that provide services such as scheduling, placement, batching, and autoscaling. Clipper [313], TensorFlow-Serving [309], MArk [310], InferLine [226], Shepherd [311], and Clockwork [224] are some earlier general ML model serving systems for serving models like ResNet that are relatively small. They support latency-aware provision to maximize the overall goodput. More recently, AlpaServe [312] utilizes model parallelism for statistical multiplexing. INFaaS [314] and Cocktail [315] propose a model-less serving framework to automate the model selection and autoscaling to meet SLOs. However, these general systems are not power-aware and fail to consider the autoregressive property of LLMs. Instead, dynamic batching and autoscaling are complementary to μ-Serve.

LLM Inference Optimization. Recently, several model-serving systems [316, 317, 318] have been proposed to optimize LLMs. Orca [316] considers the autoregressive token generation pattern of LLMs and introduces iteration-level scheduling. However, it uses a first-come-first-serve (FCFS) scheduling policy that suffers from head-of-line blocking which we address in this chapter. FastServe [317] proposes preemptive scheduling with a Multi-Level Feedback Queue. However, it introduces extra memory overhead to maintain intermediate states for unfinished jobs. vLLM [318] is a high-throughput LLM serving engine based on *PagedAttention* for token storage that achieves near-zero waste in KV cache memory. vLLM also adopts an FCFS scheduling policy for all requests. TurboTransformers [319] proposes a memory allocation algorithm to balance the memory footprint and (de-)allocation efficiency and a batching scheduler using dynamic programming to achieve optimal throughput. μ-

Serve can benefit from the memory management optimization of these systems and improve the power efficiency of serving LLMs.

Operator Placement Optimization. AlpaServe [312] partitions collections of models using inter- and intra-operator parallelism and generates operator placement strategies to reduce serving latency in the presence of bursty workloads. μ-Serve relies on AlpaServe to generate feasible operator parallelism partitioning plans. Another line of work is GPU-centric offloading, which utilizes CPU memory or disk to store portions of the model parameters. For instance, PowerInfer [343] loads frequently-activated neurons onto the GPU for fast access while offloading infrequently-activated neurons to the CPU, thus significantly reducing GPU memory demands and CPU-GPU data transfers. FlexGen [344] proposes a novel scheduling approach to prioritize throughput over latency, processing batches sequentially for each layer. DejaVu [345] accelerates LLM inference by using activation sparsity to selectively process only those neurons that are predicted to be activated. These systems are orthogonal to μ-Serve as their objective is to reduce GPU memory demand with offloading while μ-Serve aims to improve the power efficiency of model serving.

Speculative LLM Inference. Speculative decoding or look-ahead decoding accelerates LLM token generation with smaller approximation models [346, 347], multiple decoding heads [348], or n-gram generation [349]. We do not consider speculative LLM inference because of its extra computational overhead of token generation and verification.

6.6 DISCUSSION AND FUTURE CHALLENGES

Model Instance Autoscaling. Autoscaling (adding/removing model instances) is an essential orchestration task. Model replica autoscaling is another widely used model-serving technique [310, 313, 314, 315] to adapt to workload spikes. These techniques are complementary to μ-Serve. Nevertheless, we leave it to future work on integrating μ-Serve with these techniques to support dynamic model replication and request batching.

Heterogeneous Accelerators. In μ-Serve's system model, we consider only a homogeneous GPU cluster. However, GPU devices in a cluster can be heterogeneous in terms of hardware (e.g., A100, V100, and T4), resource configurations (e.g., memory size), frequency range, and power features [329, 350, 351, 352]. Device heterogeneity raises challenges in both model provisioning (e.g., which type of device to assign to a specific model partition) and dynamic frequency scaling (the power saving, idle power, and power efficiency differ across different types of devices). We leave the study of such complicated optimization space to future work.

Other Power Management Features. In μ-Serve's GPU management model, we consider only SM frequency scaling because of fast actuation and fine-grained increments. Instead, there are other power management features on more advanced GPUs such as power capping, various GPU operation modes, memory frequency scaling, and MIG sharing [324]. It is worthwhile to study the interplay between these power features and explore co-optimization policies for power saving.

Scalability. Fig. 6.7 shows that μ-Serve consistently facilitates per-device power savings as the cluster undergoes scaling out at no idle devices. Based on these results, we assert that similar power-saving opportunities are likely to be present in cloud-scale deployment while ensuring SLO attainment.

6.7 SUMMARY

In this chapter, we have presented μ-Serve, a system for serving multiple large deep learning models that maximizes power saving while preserving request-serving SLO attainment. The key innovation of μ-Serve lies in the fine-grained modeling of operator sensitivity and power-aware model provisioning that creates power-saving opportunities. Such opportunities can be exploited by dynamic GPU frequency scaling online to achieve power reduction without SLO violations at varying conditions. We quantified and discussed when and to what extent such saving opportunities exist through extensive evaluations. An open-source implementation of μ-Serve can be found at: https://gitlab.engr.illinois.edu/DEPEND/power-aware-model-serving.

CHAPTER 7: CONCLUSIONS

This dissertation has established the foundations and principles for designing practical and robust autonomous cloud infrastructures through efficient machine learning techniques. The vision is self-optimizing, self-adapting cloud utility fabrics that can continuously refine their behavior to harmonize multiple cloud efficiency objectives in a fundamentally more autonomous and robust manner, minimizing human-engineered heuristics.

This dissertation has taken the first step towards designing and managing cloud systems to optimize resource efficiency, performance isolation, and power efficiency with deep reinforcement learning. We first demonstrate that learning-based cloud systems achieve superior performance and generalizability than traditional heuristic-driven approaches in a wide range of systems and environments, including microservices resource management, serverless workload autoscaling, deep learning model serving, power management, and congestion control. Based on online telemetry data and data-driven analytics, a key advantage of these learning-based approaches is their ability to automatically tailor policies for precise deployment settings (e.g., resource contention, workload patterns, and network types). Crucially, the learned policies can dynamically adapt to challenging scenarios that fixed heuristics struggle with, especially corner or tail cases that were not explicitly accounted for during the manual heuristics design/tuning phase.

While building and deploying these learning-based systems in production cloud environments (working with our IBM collaborators), we have also identified a series of challenges that hinder the practical adoption of machine learning in systems due to issues around robustness, reliability, and the ability to maintain performant operations across all scenarios. Developing both efficient and robust learning for systems approaches fundamentally requires new machine learning algorithms with system support. In this dissertation, we have described the details of these challenges arising from cloud multi-tenancy and heterogeneity, and presented a general learning framework design with two key abstractions: (1) the ***virtual agent abstraction*** that employs mean-field theory to manage distributed learning agents in multi-tenant cloud systems, and (2) the ***meta learner abstraction*** to fast adapt learned ML models or policies across heterogeneous cloud applications and environments based on generalizable state embeddings.

The abstraction-driven design provides scalability and robustness to learning-based systems while remaining practical and highly extensible to various online learning algorithms and systems management tasks beyond resource management and power/frequency control. With the proposed framework and the open-source platform implementation, we aim to foster collaboration across research communities in machine learning, networking, and systems, and enable researchers to develop and robustly validate new machine learning approaches on real-world cloud systems management problems at scale.

While the work in this dissertation represents significant strides towards learning-based cloud systems management, many critical open challenges remain to attain this broader vision of fully autonomous cloud systems management. To conclude this dissertation, we outline several important but challenging research directions for the next steps.

7.1 LOOKING FORWARD

Policy Interpretation and Verification. For the learned policies from machine learning models to be practically adopted and deployed in mission-critical production cloud systems, it is crucial that the policies and decisions made by the ML agents are interpretable and understandable by human operators. Additionally, formal verification that the learned policies maintain certain required system properties and constraints can be beneficial before deployment. However, the black-box and unpredictable nature of modern deep neural network models presents a significant hurdle. There is understandable skepticism around applying these advanced ML techniques in systems that require high reliability, as even well-trained neural networks can sometimes lead to suboptimal or catastrophic outcomes in corner cases [252, 353] or be exploited by adversarial attacks [354].

While progress has been made in enhancing ML explainability, such as techniques to translate neural network policies into interpretable rule-based models like decision trees [355, 356, 357], key challenges remain. Maintaining the performance benefits of deep RL policies while providing full transparency is difficult. In formal verification of machine learning for systems [358, 359], the complexity increases exponentially with the size of the neural network and environment state space. Moreover, it is challenging to comprehensively specify the system properties that must be satisfied for each cloud management task.

Overcoming these hurdles by combining model interpretation, verification, and controlled deployment approaches could pave the way for the safe adoption of learned policies in production. One potential approach is a two-stage workflow: first performing policy exploration, training, and rule extraction in a sandboxed environment. Then the interpretable, verified rule-based policies can be deployed at scale on production cloud infrastructure. Addressing these challenges through fundamental innovations in ML explainability, verification, and their integration into the learned systems management frameworks will be critical to realizing the vision of robust, trustworthy autonomous cloud control planes. Collaborations spanning machine learning, formal methods, cloud systems, and operational practices will be vital to make pragmatic progress.

Foundation Model for Systems. Lately, foundation models (FMs) have exhibited notable efficacy in tackling a range of intricate tasks, especially in natural language processing (NLP). 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 [18]. FMs have been driving a paradigm shift in the way that modern-day ML models are trained and used. 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 [360] or few/zero-shot learning [361, 362].

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 or in the area of systems and networking in general. The existing way of developing and training one model per systems task is costly in terms of both training time and dataset collection effort [58]. In this dissertation, we have shown that the learned model in each task requires substantial retraining (even with transfer learning) when serving new applications or cloud environments. FLASH [363, 364] facilitates model adaptability across applications or environments (but not across tasks). Ultimately, we desire an FM that can learn general systems management policies across tasks [305]. 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.

Combined Systems and ML Resilience. The widespread adoption of learning-based system agents, or *ML agents* in short, across the various system stacks and layers of cloud datacenters has introduced a new dimension to the challenge of ensuring system resilience. Traditionally, system resilience focused on mitigating faults and failures within the hardware and software components, employing techniques such as redundancy, failover mechanisms, and error handling [365]. However, with the proliferation of ML agents making predictions and generating system management or control decisions, the fault model has expanded beyond the traditional realm.

These ML agents, whether employing supervised learning, RL, or recent advancements like general or fine-tuned large language models (LLMs), introduce a new set of potential failure modes. ML models, despite their sophistication, can be susceptible to data/model uncertainties, biases, out-of-training-distribution shifts, adversarial attacks, and other forms of degradation or unexpected behavior (e.g., silent data corruptions [366]). Consequently, the requirements for resilient systems must now encompass the resilience of these ML agents themselves. There is an urgent need to understand and improve the systems and ML resilience in large-scale cloud datacenters. One promising direction could be to combine mechanistic models [367] that capture domain knowledge with data-driven learning approaches, leveraging the strengths of both to build more robust and reliable systems.

Sustainable Green Computing. Today's hyper-scale cloud datacenters consume significant power. It has been reported that they already contribute 2–3% of the overall global carbon footprint and will account for 8% by 2030 [368]. Meanwhile, emerging deep learning and large model training or inference workloads [18, 369, 370]) are demanding increasing amounts of power. Achieving carbon efficiency for a sustainable future is a daunting challenge. Learning-based cloud systems management must address the urgent need to transition to green computing while retaining the ability to meet joint performance and resilience objectives. The goal is to reduce the carbon footprint by optimizing the usage of green energy sources (e.g., solar energy), which is desirable but costly and relatively unstable while continuously meeting cloud service SLOs in availability and performance. The major challenges to achieving that goal are outlined below:

- *Fundamental Trade-off between Sustainability and Cloud SLOs.* Sustainable computing requires both sustainable energy costs and sustainable cloud operations (by meeting SLOs). Conversely, meeting stringent SLOs can incur high energy costs (e.g., due to overprovisioning). Cloud datacenters require careful design and optimization in dealing with this fundamental trade-off.

- *Variability in Green Energy Supply and Dynamic Workload.* Green energy sources are inherently unstable [371], and cloud datacenter workloads also exhibit dynamically varying spatial and temporal patterns. This requires a continuously optimized trade-off between cloud SLO violations and carbon emissions, posing a challenging multi-objective optimization problem.

- *Lack of an Application-aware Power Control Plane.* Existing efforts adopt a *top-down* approach in minimizing carbon footprint, such as load shifting based on predictions of carbon intensity [372]. However, conservative power control misses energy-saving opportunities, while application-agnostic aggressive power control can lead to SLO violations [252]. Therefore, it is necessary to take a scalable, application-aware [373], *bottom-up* optimization approach and incorporate hardware/software co-design for today's hyper-scale cloud datacenters.

APPENDIX A: META LEARNING CASE STUDIES

A.1 META LEARNER ARCHITECTURE IN FLASH

In this section, we introduce the neural network architecture of FLASH's meta learner (as shown in Fig. A.1) 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 Section 5.6.1, 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 the t-th sample, i.e., $X_t = [x_t, y_t]$. 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

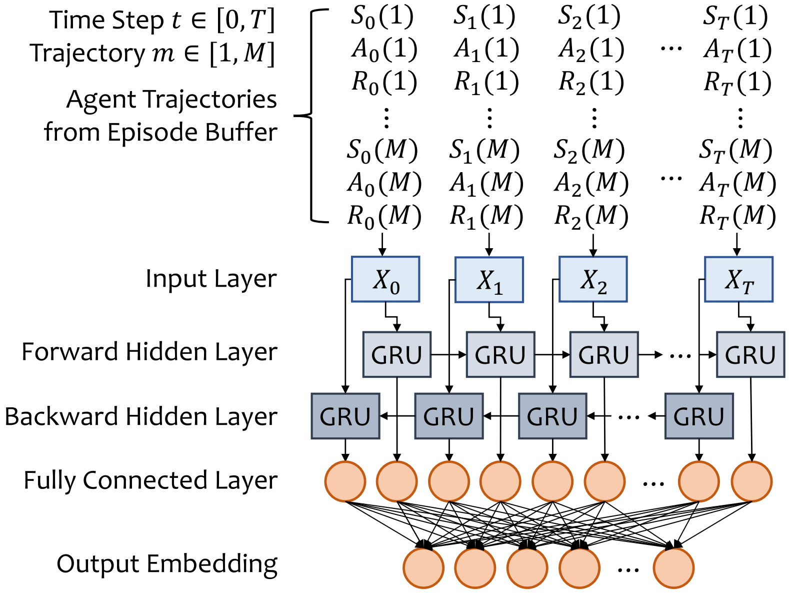

Time Step $t \in [0, T]$
Trajectory $m \in [1, M]$
Agent Trajectories from Episode Buffer
$S_0(1)$ $A_0(1)$ $R_0(1)$
$S_1(1)$ $A_1(1)$ $R_1(1)$
$S_2(1)$ $A_2(1)$ $R_2(1)$
$S_T(1)$ $A_T(1)$ $R_T(1)$
$S_0(M)$ $A_0(M)$ $R_0(M)$
$S_1(M)$ $A_1(M)$ $R_1(M)$
$S_2(M)$ $A_2(M)$ $R_2(M)$
$S_T(M)$ $A_T(M)$ $R_T(M)$
Input Layer
X_0
X_1
X_2
X_T
Forward Hidden Layer
GRU
Backward Hidden Layer
GRU
Fully Connected Layer
Output Embedding

Figure A.1: Neural network architecture of FLASH's RNN-based meta learner for embedding generation.

$t \in [0, T]$ where T is the trajectory length. For instance, as shown in Fig. A.1, X_t contains $[S_t(m), A_t(m), R_t(m)]$ 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 Section 5.6.1, FLASH uses a bidirectional GRU (Gated Recurrent Unit), a special class of RNN [275], 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 [374] to the input sequences. Two unidirectional RNN hidden layers are chained together in opposite directions and act on the same input (as shown in Fig. A.1). 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.

We do not explicitly use the popular memory augmentation technique [276] for the meta learner as the features of our application workloads are not as high-dimensional as those of computer vision tasks [276]. We also leave the usage of more advanced sequence models such as long short-term memory (LSTM) [375, 376] and attention-based techniques (e.g., Transformers [301]) 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 [139], and the hyperparameters are shown in Table A.1.

Table A.1: 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

A.2 DETAILS OF CASE STUDY ON SIZELESS

Model. Sizeless [64] uses a fully connected neural network as the predictor for the regression task. The features (after feature engineering) and label used in the Sizeless model are shown in Table A.2, which are consistent with the original paper [64]. After a grid search to tune the hyperparameters of the model (as shown in Table A.3), 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 [64] based on a survey of 89 industry use cases of serverless computing applications [266]. 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 [64] 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 [64]. 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 A.2: 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 A.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

OpenWhisk Dataset. Following the same dataset collection methodology as Sizeless [64], 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 [235] 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 Section 2.4 and Section 5.6.3.

A.3 DETAILS OF CASE STUDY ON FIRM

Model. As described in Section 3.2, FIRM [1] uses an actor-critic RL algorithm, DDPG [210]. The RL agent monitors the system- and application-specific measurements and learns how to scale the allocated resources vertically and horizontally. Table 3.2 shows the agent's state and action spaces. Table 3.3 shows the model hyperparameters. 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, $SP = 1$. 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 [267] 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).

A.4 DETAILS OF CASE STUDY ON SMARTOVERCLOCK

Model. SmartOverclock [16] 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 [211]. 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 A.4 shows the model's state and action space. Table A.5 shows the model training hyperparameters.

Table A.4: RL state-action space and reward function in SmartOverclock [16] 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)
$\alpha \cdot ((IPS_t - IPS_{t-1})/IPS_t)_{\Delta freq>0} + (1-\alpha) \cdot SP_t$

Table A.5: SmartOverclock (Q-Learning) training hyperparameters (adopted from Stable Baseline3 [211]).

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

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 frequency 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 [71, 72, 74, 254] has proposed RL-based solutions. For instance, Aurora [72] 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 [72].

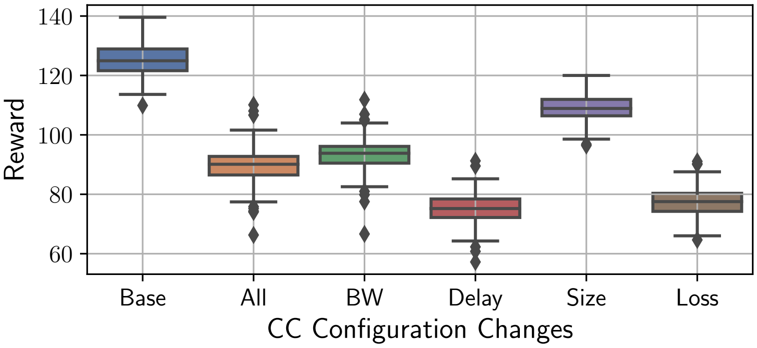


Figure A.2: Reward degradation in Aurora.

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 [66, 70, 72] (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. 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. Fig. A.2 shows the per-episode reward of the Aurora agent in different scenarios of network environment changes. The 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 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 [16, 75, 76, 77, 285] where the agent adjusts the frequency of CPU cores to balance application performance improvements with the extra power cost. For example, SmartOverclock [16] 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. Fig. A.3 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.

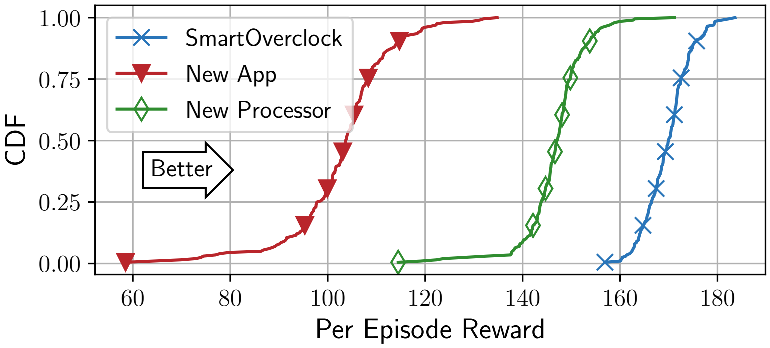


Figure A.3: Reward degradation in SmartOverclock.

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. 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 [72], 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 [72] uses an actor-critic RL algorithm, PPO [209]. Table A.6 shows the state and action spaces of the RL model. Table A.7 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 and SmartOverclock (as described in Section 5.6.2), 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.

Evaluation Setup. We train and test FLASH-Aurora in a network environment simulator [72] that can simulate network links with a wide variety of network scenarios. The

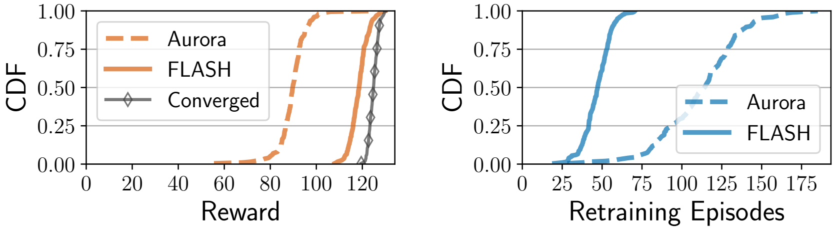
Table A.6: RL state-action space and reward function in Autora [72] for the congestion control task.

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 A.7: Aurora training hyperparameters (PPO algorithm).

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

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 Fig. 5.14), and the adaptation evaluation is done on the remaining 425 environments (i.e., "adaptation pool").



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

Figure A.4: Comparison of the performance and retraining cost of the Aurora model with FLASH.

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 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). Fig. A.4(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 Fig. A.4(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 Fig. A.4(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 Section 5.8, 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 pretraining pool and then used for adaptation to each novel 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 Fig. A.5 the training cost analysis of FLASH and TL with different sizes of the

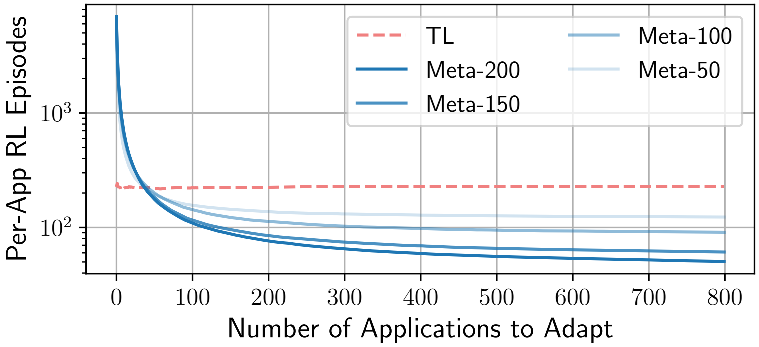


Figure A.5: Training cost analysis of transfer learning (TL) and meta-learning with various sizes of pretraining application pool in the task of workload autoscaling.

pretraining application pool in the task of workload autoscaling. The X-axis represents the number of applications that the model has been adapted to, and the Y-axis (in log-scale) 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 Fig. A.5), 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.

APPENDIX B: ADDITIONAL EXPERIMENTS ON μ-SERVE

B.1 ABLATION STUDY ON PROXY MODELS

Model-Specific Results. In Fig. B.1, we plot the detailed evaluation results of our output token length predictor on each individual model from the LMSYS-Chat-1M dataset. We consider standard evaluation metrics including accuracy, precision, recall, and F1 score. Our predictor achieves rather balanced performances across different models, with the prediction accuracy ranging from 0.33 for `llama-13b` to 0.50 for `wizardlm-13b`.

Evaluation Results for Smaller Predictor Models. In the main text of the paper, we primarily build our predictor on top of the "BERT-Base" model [377] with 12 Transformer layers and 768 hidden dimensions. In this appendix, we consider a smaller predictor based on the "BERT-Tiny" model with 2 layers and 128 dimensions. Table B.1 summarizes the evaluation results of multiple predictors built upon BERT-Tiny using different training schemes, including ordinal classification, standard classification, and regression. Compared with the BERT-Base results in Table 6.2, we can see that the smaller BERT-Tiny model has lower prediction performance in almost every metric that we consider. In fact, the BERT-Tiny-based regressors (trained using MSE or L1 loss) seem to significantly underfit the training data and do not perform much better than random guesses. This is expected because the smaller BERT-Tiny architecture is less expressive and can hardly capture the subtle features from the input queries in natural languages.

Scheduling Performance Comparison In terms of the prediction latency overhead, the average, p99.9, and maximum inference latencies of the predictor built on top of BERT-Tiny are 1.8 ms, 3.7 ms, and 10.2 ms, respectively, less than that of the predictor built with

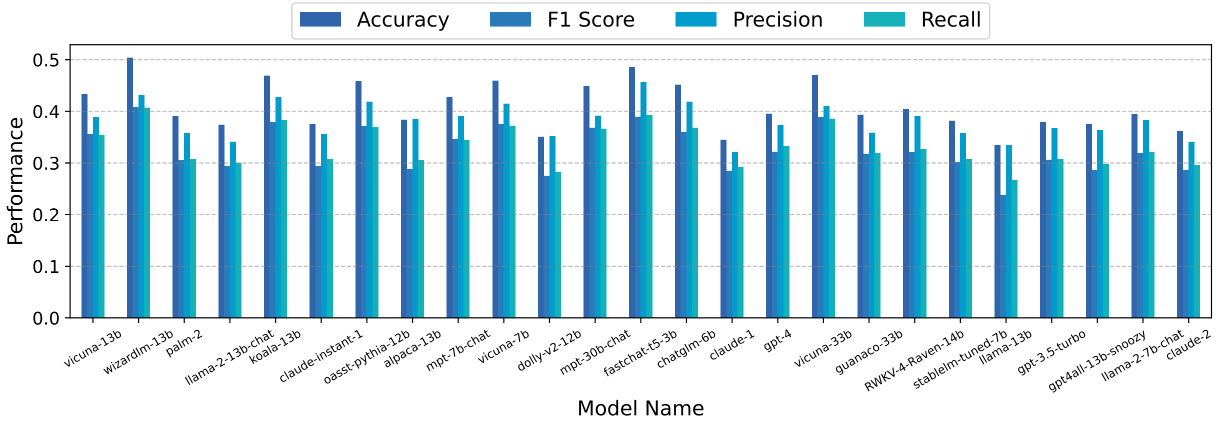


Figure B.1: Output token length predictor evaluation results for each individual model.

Table B.1: BERT-tiny-based output token length predictor evaluation results.

Metrics	Ord. CLS (MSE)	Ord. CLS (L1)	CLS	REG (MSE)	REG (L1)
Accuracy ↑	0.3947	0.4097	**0.4193**	0.2425	0.2425
F1 Score ↑	0.3121	0.3193	**0.3687**	0.0781	0.0781
L1 Error ↓	0.7752	**0.7558**	N/A	0.9528	0.9530
MSE ↓	**0.9656**	1.0398	N/A	1.8639	1.8643

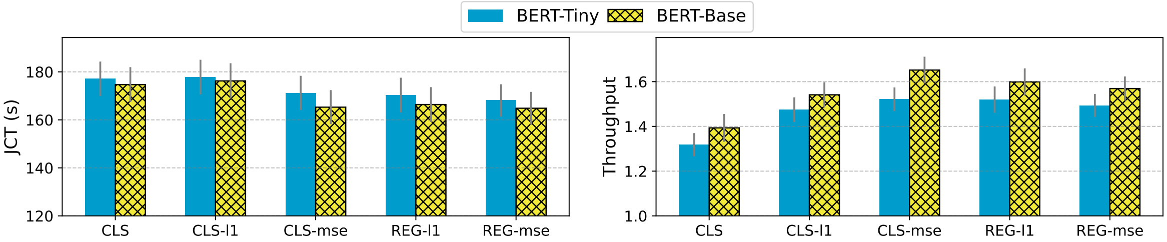


Figure B.2: Scheduling performance comparison with BERT-Base and BERT-Tiny as the proxy models.

BERT-Base. We then use predictors based on BERT-Base and BERT-Tiny proxy models in the μ-Serve scheduler and compare the scheduling performance regarding the job completion time (JCT) and throughput. Fig. B.2 shows the comparison of the two proxy models when we use different training schemes: multi-class classification, ordinal classification with L1 loss and MSE, and regression with L1 loss and MSE. The scheduler with BERT-Base as the proxy model achieves 1–3.6% less JCT and 4.7–8.5% higher throughput. As expected, the smaller model incurs even less latency overhead but suffers worse prediction accuracy and scheduling performance in return. Therefore, we use BERT-Base as the proxy model in μ-Serve's scheduler.

B.2 ABLATION STUDY ON μ-SERVE SCHEDULER

In this section, we conduct experiments to understand the role of the speculative shortest-job-first (SJF) scheduler (Section 6.3.3) in power saving. The results in Section 6.4.3 show an improvement of 38.6–41.3% in JCT and 2.2–2.5× in throughput. Using the same setup as in Section 6.4.2, we compare three methods: AlpaServe, AlpaServe-Scale (No Sched), and μ-Serve (No Sched). All three methods are implemented without μ-Serve scheduler, i.e., they are based on the default FCFS scheduler in AlpaServe. Fig. B.3 and Fig. B.4 show the comparison of power saving and SLO attainment of the three methods.

In terms of SLO attainment, without μ-Serve scheduler, there are more situations that the p99 latency is above the SLO latency as suggested by the wider red-shaded area compared

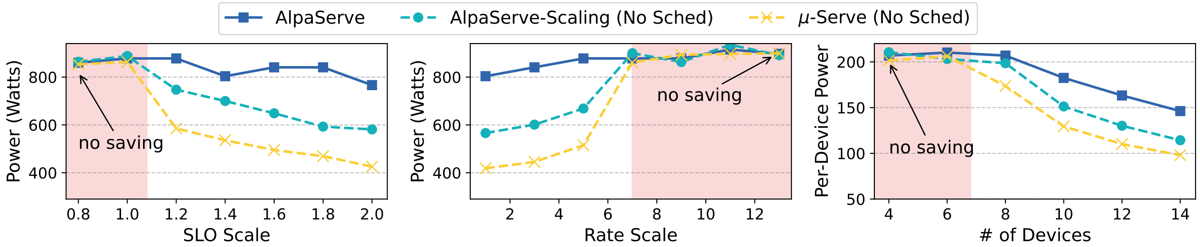


Figure B.3: Power saving evaluation (without μ-Serve scheduler).

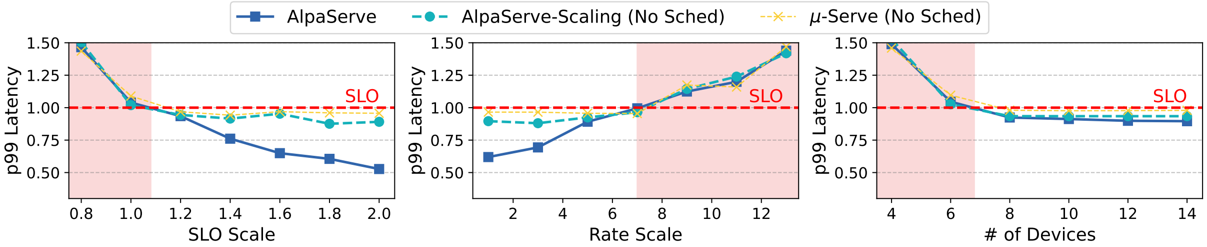


Figure B.4: SLO preservation evaluation (without μ-Serve scheduler).

to Fig. 6.8 (indicating worse SLO attainment). When there is an SLO attainment violation, online GPU device frequency scaling does not actuate any down-scaling. Even without SLO attainment violation, AlpaServe has a worse SLO attainment compared with AlpaServe-Sched (up to 31% lower p99 latency as shown in Fig. 6.8). Therefore, the power-saving opportunities are reduced.

Fig. B.3 shows the power consumption when running the three approaches. At SLO attainment violations (i.e., in the red-shaded area), there is no power saving. When there are no SLO attainment violations, the power saving achieved by μ-Serve (No Sched) compared to AlpaServe is up to 1.8×, 1.92×, and 1.49× at varying SLO scales, rates, and numbers of devices (less than the improvement of 2.3×, 2.61×, and 2× when μ-Serve runs on its scheduler as shown in Fig. 6.7). This 21–27% reduction in power saving attributed to the missing of μ-Serve's scheduler justifies the role of the scheduler in μ-Serve empirically. AlpaServe-Scaling (No Sched) has up to 1.42×, 1.42×, and 1.28× power saving compared to AlpaServe at varying SLO scales, rates, and numbers of devices, respectively. Compared to the improvement of AlpaServe-Scaling on top of AlpaServe (as shown in Fig. 6.7), AlpaServe-Scaling (No Sched) also has less improvement (e.g., 1.42× $<$ 1.7× improvement at varying rates).

Without the speculative SJF scheduler in μ-Serve, the power saving from μ-Serve and AlpaServe-Scaling becomes 21–27% less. The main reason is that replacing μ-Serve's scheduler with FCFS leads to worse SLO attainment and more SLO violations, and consequently less GPU frequency down-scaling (i.e., power saving) opportunities.

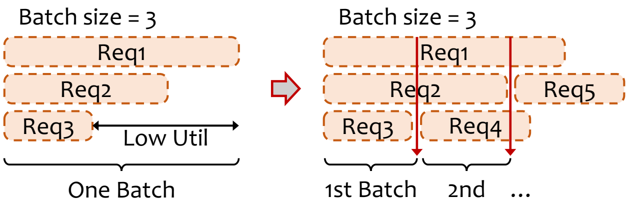
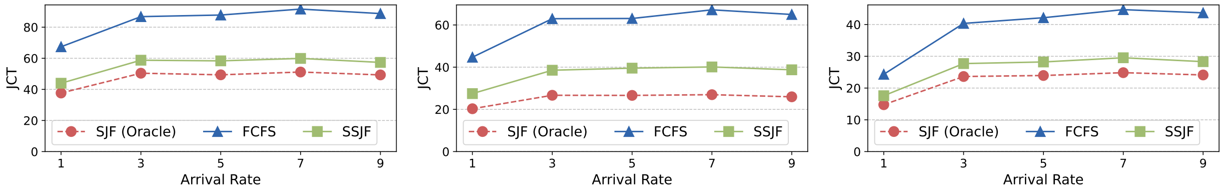
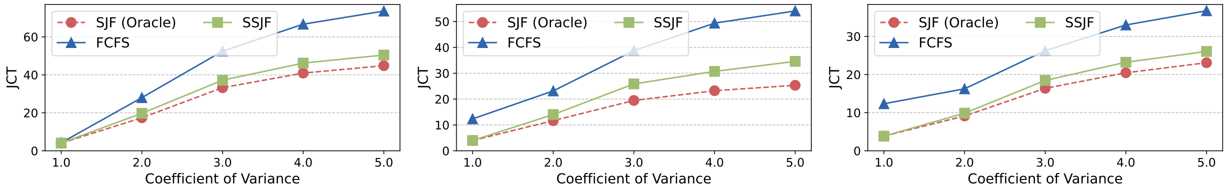


Figure B.5: Batching vs. Continuous batching.



(a) No batching. **(b)** Dynamic batching. **(c)** Continuous batching.

Figure B.6: Job completion time (JCT) with varying rates.



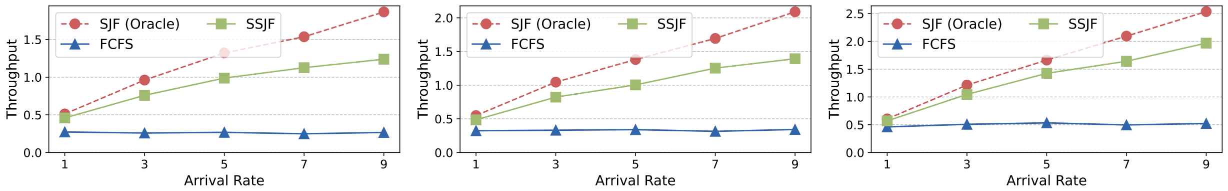
(a) No batching. **(b)** Dynamic batching. **(c)** Continuous batching.

Figure B.7: Job completion time (JCT) with varying burstiness.

B.3 SUPPORT FOR VARIOUS BATCHING TECHNIQUES

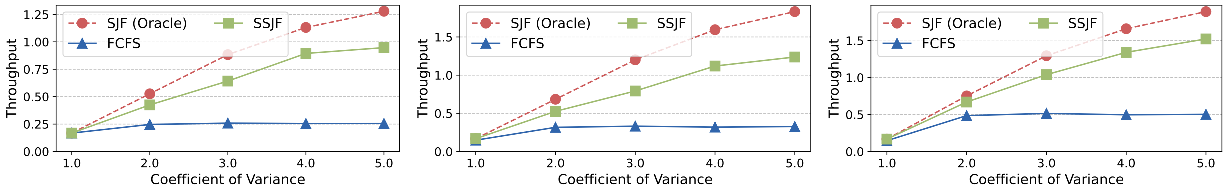
With predictions of the output token length of each incoming inference request, μ-Serve's scheduler schedules requests with shorter predicted lengths to run first. μ-Serve supports three modes of batching: no batching, dynamic batching, and continuous batching. In the no-batching mode, the request is served one by one according to the execution order determined by μ-Serve's scheduler. In the dynamic-batching mode, the scheduler waits for `batch_wait_timeout` or until `max_batch_size` requests in each batch have been filled up whose order is determined by μ-Serve. In the continuous-batching mode (first proposed by Orca [316]), requests are batched at the iteration level and the early-finished request is substituted by the next request in the waiting queue (ordered by μ-Serve). The difference between (dynamic) batching and continuous batching is illustrated in Fig. B.5.

We evaluate the scheduling performance regarding the job completion time (JCT) and the serving throughput to understand if prediction accuracy is sufficient. Our comparison baselines include (1) first-come-first-serve (FCFS), which is the default scheduler in state-



(a) No batching. **(b)** Dynamic batching. **(c)** Continuous batching.

Figure B.8: Throughput with varying rates.



(a) No batching. **(b)** Dynamic batching. **(c)** Continuous batching.

Figure B.9: Throughput with varying burstiness.

of-the-art model-serving frameworks such as Orca [316] and vLLM [318], and (2) shortest-job-first (SJF) with the actual output token length (i.e., oracle).

Request Serving JCT. We evaluate μ-Serve and the comparison baselines in three batching settings: (1) no batching, (2) dynamic batching, and (3) continuous batching (max batch size is set to 4 for both batching cases). Fig. B.6 and Fig. B.7 show the average LLM serving JCT with varying request arrival rates and burstiness. With varying request rates, μ-Serve reduces JCT by 34.5%, 39.6%, and 33.2% compared to the FCFS scheduler, under three batching settings respectively. In comparison, the oracle SJF scheduler reduces the JCT by 43.7%, 58.2%, and 43.0%. Under different burstinesses, μ-Serve reduces JCT by 30.5%, 39.0%, 35.0% compared to FCFS, while the oracle SJF reduces the JCT by 37.6%, 52.9%, and 41.5%.

Request Serving Throughput. Under the same experimental settings, we present the throughput results in Fig. B.8 and Fig. B.9. With varying request rates, μ-Serve increases the throughput by 3.6×, 3.0×, and 2.8× in the no batching, dynamic batching, and continuous batching cases, respectively. In comparison, the oracle SJF increases the throughput by 4.7×, 4.1×, and 3.2×. Under different burstinesses, μ-Serve increases the throughput by 2.6×, 2.6×, and 2.2× while the oracle SJF increases the throughput by 3.4×, 3.8×, and 2.7×.

At Varying Batch Sizes. We repeat the experiments while increasing the batch sizes. Results are shown in Fig. B.10. Continuous batching always has less JCT and higher throughput than dynamic batching (same results as shown in [316]). At varying batch sizes, μ-Serve's

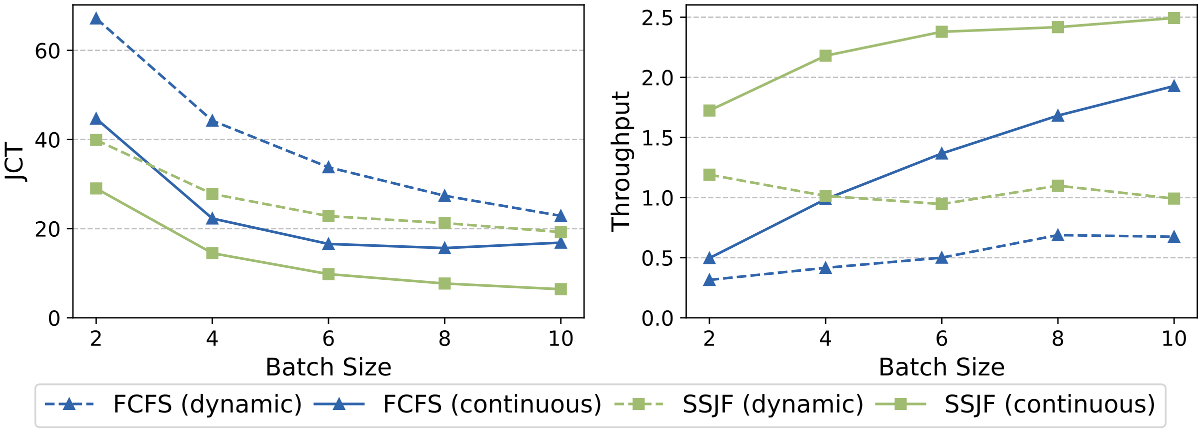


Figure B.10: Benefits of SSJF at varying batch sizes.

scheduler performance is better than FCFS's but the improvements in JCT and throughput that μ-Serve brings decrease as the batch size increases. This is because the larger the batch, the less difference there is when reordering the queue.

B.4 SUPPORT FOR MULTI-ROUND LLM CONVERSATIONS.

In addition to general LLM inferences, μ-Serve's scheduler also supports interactive LLM inference that happens in rounds. Previous rounds of user input queries and LLM responses are usually provided as *context* to the current-round input query. However, the input limit to BERT-base as a proxy model is 512 tokens, which can be much less than the context window. To support multi-round conversations, we use a simple strategy of concatenation and truncation. Specifically, we concatenate all the previous-round user prompts (P1, P2, P3, etc.), LLM responses (R1, R2, R3, etc.), and the current-round user prompt to form a context-augmented input query for the current round of conversation. Then, we retain only the last 512 tokens of the concatenated query, dropping the earlier portions. By concatenating the previous conversation history and truncating it if necessary, this approach allows a proxy model to effectively handle multi-round conversations while staying within the input length limitation.

At Different Conversation Rounds. We evaluate the effectiveness of μ-Serve's support for multi-round LLM conversations. Results are shown in Fig. B.11. At different conversation rounds, the improvement of JCT (compared to FCFS) varies from 33.1% (at round #5) to 38.9% (at round #1); The improvement of throughput varies from 1.67× (at round #5) to 2.22× (at round #1).

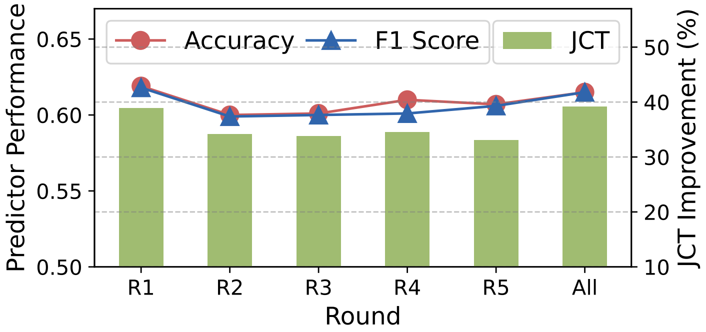


Figure B.11: SSJF evaluation across conversation rounds.

B.5 POTENTIAL USES OF μ-SERVE SCHEDULER FOR OTHER TASKS.

In Chapter 6, we study the benefit of μ-Serve's scheduler in model-serving latency and throughput, and consequently the SLO attainment. Because of higher SLO attainment, there is more headroom for dynamically down-scaling GPU device frequencies to save power without introducing SLO attainment violations. However, we anticipate that there can be more potential use cases and extensions of μ-Serve's proxy-model-based scheduler.

- *Optimization in model-serving latency or throughput.* A proxy-model-based model output length predictor can be used with existing model-serving systems (e.g., Clipper [313], TensorFlow-Serving [309], MArk [310], InferLine [226], Shepherd [311], and Clockwork [224]) to maximize the systems throughput/goodput or to support better latency-aware provision by avoiding head-of-line blocking and optimizing the waiting time of queued requests.

- *Optimized batching and GPU utilization.* Output token length prediction can be used to batch requests with similar execution lengths together to improve GPU utilization and prevent shorter requests from waiting for longer ones [316].

- *Optimization in model-serving memory and key-value cache management.* The memory management system needs to accommodate a wide range of output lengths [318]. In addition, as the output length of a request grows at decoding, the memory required for its key-value cache also expands and may exhaust available memory for incoming requests or ongoing generation for existing requests. μ-Serve's prediction can potentially help make scheduling decisions, such as deleting or swapping out the key-value cache of some requests from GPU memory.

REFERENCES

[1] 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)*. Berkeley, CA, USA: USENIX Association, Nov. 2020, pp. 805–825.

[2] 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)*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 306–322.

[3] H. Qiu, W. Mao, A. Patke, S. Cui, S. Jha, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "Power-aware deep learning model serving with μ-serve," in *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC 2024)*, 2024.

[4] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, p. 50–57, 4 2016. [Online]. Available: https://doi.org/10.1145/2890784

[5] L. A. Barroso and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pp. 1–108, 2009.

[6] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "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)*. New York, NY, USA: Association for Computing Machinery, 2019, p. 19–33.

[7] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[8] C. Delimitrou and C. Kozyrakis, "QoS-aware scheduling in heterogeneous datacenters with Paragon," *ACM Transactions Computer Systems*, vol. 31, no. 4, 12 2013. [Online]. Available: https://doi.org/10.1145/2556583

[9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* Cambridge, MA, USA: MIT Press, 2018.

[10] G. Papoudakis, F. Christianos, A. Rahman, and S. V. Albrecht, "Dealing with non-stationarity in multi-agent deep reinforcement learning," 2019. [Online]. Available: https://arxiv.org/abs/1906.04737

[11] A. Xie, J. Harrison, and C. Finn, "Deep reinforcement learning amidst lifelong non-stationarity," in *Proceedings of the 38th International Conference on Machine Learning (ICML 2021).* Cambridge, MA: PMLR, 7 2021, pp. 1–11.

[12] C. Claus and C. Boutilier, "The dynamics of reinforcement learning in cooperative multi-agent systems," *Proceedings of the 15th National Conference on Artificial Intelligence and the 10th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 1998)*, vol. 2, pp. 746–752, 1998.

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

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

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

[16] 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).* New York, NY, USA: Association for Computing Machinery, 2022, pp. 622–634.

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

[18] 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.

[19] C.-J. Wu, R. Raghavendra, U. Gupta, B. Acun, N. Ardalani, K. Maeng, G. Chang, F. Aga, J. Huang, C. Bai et al., "Sustainable AI: Environmental implications, challenges, and opportunities," *Proceedings of the 5th Annual Conference on Machine Learning and Systems (MLSys 2022)*, vol. 4, pp. 795–813, 2022.

[20] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving low tail latency for microsecond-scale networked tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017)*, 2017, pp. 325–341.

[21] C. Satnic, "Amazon, Microservices and the birth of AWS cloud computing," Apr. 2016, https://www.linkedin.com/pulse/amazon-microservices-birth-aws-cloud-computing-cristian-satnic/, Accessed 2020/01/23.

[22] D. Paik, "Adapt or Die: A microservices story at Google," Dec. 2016, https://www.slideshare.net/apigee/adapt-or-die-a-microservices-story-at-google, Accessed 2020/01/23.

[23] S. Ihde and K. Parikh, "From a monolith to microservices + REST: The evolution of LinkedIn's service architecture," Mar. 2015, https://www.infoq.com/presentations/linkedin-microservices-urn/, Accessed 2020/01/23.

[24] T. Mauro, "Adopting microservices at Netflix: Lessons for architectural design," Feb. 2015, https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, Accessed 2020/01/23.

[25] B. Zoph, C. Raffel, D. Schuurmans, D. Yogatama, D. Zhou, D. Metzler, E. H. Chi, J. Wei, J. Dean, L. B. Fedus, M. P. Bosma, O. Vinyals, P. Liang, S. Borgeaud, T. B. Hashimoto, and Y. Tay, "Emergent abilities of large language models," *Transactions on Machine Learning Research*, 2022.

[26] S. Chasins, A. Cheung, N. Crooks, A. Ghodsi, K. Goldberg, J. E. Gonzalez, J. M. Hellerstein, M. I. Jordan, A. D. Joseph, M. W. Mahoney et al., "The sky above the clouds," *arXiv preprint arXiv:2205.07147*, 2022.

[27] S. Senthil Kumaran, *Practical LXC and LXD: Linux Containers for Virtualization and Orchestration*. Springer, 2017.

[28] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, pp. 2–2, 2014.

[29] "CoreOS rkt, a security-minded, standards-based container engine," https://coreos.com/rkt/, Accessed 2020/01/23.

[30] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS containers," in *Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, 2018, pp. 199–212.

[31] "Docker Swarm," https://www.docker.com/products/docker-swarm, Accessed 2020/01/23.

[32] "Kubernetes," https://kubernetes.io/, Accessed 2020/01/23.

[33] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys 2013)*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 351–364.

[34] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the 10th European Conference on Computer Systems (EuroSys 2015).* New York, NY, USA: Association for Computing Machinery, 2015, pp. 1–17.

[35] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to cloud-native architectures using microservices: An experience report," in *Proceedings of the European Conference on Service-Oriented and Cloud Computing.* Springer, 2015, pp. 201–215.

[36] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[37] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.

[38] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 155–158, 2018.

[39] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024)*, 2019, pp. 3–18.

[40] K. G. Lockyer, *Introduction to Critical Path Analysis.* Pitman, 1969.

[41] C.-Q. Yang and B. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS 1988)*, 1988, pp. 366–367.

[42] "Train-Ticket: A train-ticket booking system based on microservice architecture," https://github.com/FudanSELab/train-ticket, Accessed 2020/01/23.

[43] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren, "Cloud control with distributed rate limiting," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 337–348, 2007.

[44] B. Ager, F. Schneider, J. Kim, and A. Feldmann, "Revisiting cacheability in times of user generated content," in *2010 INFOCOM IEEE Conference on Computer Communications Workshops.* IEEE, 2010, pp. 1–6.

[45] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscaling policies for complex workflows," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 75–86.

[46] Y. Ahn, J. Choi, S. Jeong, and Y. Kim, "Auto-scaling method in hybrid cloud for scientific applications," in *Proceedings of the 16th Asia-Pacific Network Operations and Management Symposium*. IEEE, 2014, pp. 1–4.

[47] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018.

[48] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[49] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *Proceedings of 2019 IEEE International Conference on Web Services (ICWS 2019)*. IEEE, 2019, pp. 68–75.

[50] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *Proceedings of 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS 2019)*. IEEE, 2019, pp. 1994–2004.

[51] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*, 2018, pp. 783–798.

[52] S. Stüdli, M. Corless, R. H. Middleton, and R. Shorten, "On the modified AIMD algorithm for distributed resource management with saturation of each user's share," in *Proceedings of 2015 54th IEEE Conference on Decision and Control (CDC 2015)*. IEEE, 2015, pp. 1631–1636.

[53] P. Gevros and J. Crowcroft, "Distributed resource management with heterogeneous linear controls," *Computer Networks*, vol. 45, no. 6, pp. 835–858, 2004.

[54] J. Dejun, G. Pierre, and C.-H. Chi, "Resource provisioning of web applications in heterogeneous clouds," in *Proceedings of the 2nd USENIX Conference on Web Application Development*. USENIX Association, 2011, pp. 49–60.

[55] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *Proceedings of 2011 31st International Conference on Distributed Computing Systems*. IEEE, 2011, pp. 559–570.

[56] 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*, vol. 63, no. 2, p. 50–59, jan 2020.

[57] 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: USENIX Association, 2020.

[58] 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).* Boston, MA: USENIX Association, Apr. 2023, pp. 1097–1114.

[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).* New York, NY, USA: Association for Computing Machinery, 2022, pp. 16–30.

[60] 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).* USA: USENIX Association, 2016, p. 363–378.

[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).* New York, NY, USA: Association for Computing Machinery, 2017, p. 452–465.

[62] 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).* USA: USENIX Association, 2018, p. 759–773.

[63] Z. Yanqi, H. Weizhe, Z. Zhuangzhuang, S. G. Edward, and D. Christina, "Sinan: ML-based & QoS-aware resource management for cloud microservices," in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021).* New York, NY, USA: Association for Computing Machinery, 2021, p. 167–181.

[64] 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).* Association for Computing Machinery, 2021, p. 248–259.

[65] 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).* New York, NY, USA: Association for Computing Machinery, 2016, pp. 50–56.

[66] 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 (NIPS 2019)*, vol. 32, 2019.

[67] 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)*. Cambridge, MA, USA: PMLR, 2020, pp. 629–641.

[68] 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)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 67–74.

[69] 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)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 270–288.

[70] 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*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 397–413.

[71] 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)*, vol. 36, no. 11, pp. 12 615–12 621, 6 2022.

[72] N. Jay, N. H. Rotman, P. Godfrey, M. Schapira, and A. Tamar, "Internet congestion control via deep reinforcement learning," *In Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS 2018)*, vol. 32, 2018.

[73] 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)*. USENIX Association, 2021, pp. 611–624.

[74] 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)*. New York, NY, USA: Association for Computing Machinery, 2022, p. 261–275.

[75] 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)*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 445–460.

[76] 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*, vol. 13, no. 3, pp. 146–149, 2021.

[77] F. M. M. u. Islam and M. Lin, "Hybrid DVFS scheduling for real-time systems based on reinforcement learning," *IEEE Systems Journal*, vol. 11, no. 2, pp. 931–940, 2017.

[78] 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 2022)*. New York, NY, USA: Association for Computing Machinery, 2022, p. 20–28.

[79] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI 2018)*. USA: USENIX Association, 2018, pp. 561–577.

[80] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2018, pp. 133–146.

[81] D. A. Patterson, "Technical perspective: The data center is the computer," *Communications of the ACM*, vol. 51, p. 105, 2008.

[82] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing transient congestion during network update in data centers," in *Proceedings of the 2014 Conference on Emerging Networking EXperiments and Technologies*, 2014, pp. 4–6.

[83] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "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)*. New York, NY, USA: Association for Computing Machinery, 2015, p. 183–197.

[84] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," *SIGARCH Computer Architecture News*, vol. 43, no. 3S, p. 158–169, jun 2015.

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

[86] L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta, "Robust adversarial reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*. JMLR.org, 2017, pp. 2817–2826.

[87] J. Morimoto and K. Doya, "Robust reinforcement learning," in *Advances in Neural Information Processing Systems (NeurIPS 2000)*, T. Leen, T. Dietterich, and V. Tresp, Eds., vol. 13. MIT Press, 2000, https://proceedings.neurips.cc/paper/2000.

[88] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. Winstein, "Learning in situ: A randomized experiment in video streaming," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2020)*, 2020, pp. 495–511.

[89] J. Ding, R. Cao, I. Saravanan, N. Morris, and C. Stewart, "Characterizing service level objectives for cloud services: Realities and myths," in *Proceedings of 2019 IEEE International Conference on Autonomic Computing (ICAC 2019)*. IEEE, 2019, pp. 200–206.

[90] A. Sriraman and T. F. Wenisch, "μSuite: a benchmark suite for microservices," in *Proceedings of the 2018 IEEE International Symposium on Workload Characterization (IISWC 2018)*. IEEE, 2018, pp. 1–12.

[91] A. Sriraman and T. F. Wenisch, "μtune: Auto-tuned threading for OLDI microservices," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*, 2018, pp. 177–194.

[92] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019, pp. 25–32.

[93] I. Arapakis, X. Bai, and B. B. Cambazoglu, "Impact of response latency on user behavior in web search," in *Proceedings of The 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, 2014, pp. 103–112.

[94] T. Hoff, "Latency is everywhere and it costs you sales: How to crush it," July 2009, http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it, Accessed 2020/01/23.

[95] M. McGee, "It's official: Google now counts site speed as a ranking factor," Apr. 2010, https://searchengineland.com/google-now-counts-site-speed-as-ranking-factor-39708, Accessed 2020/01/23.

[96] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC 2012)*, 2012, pp. 1–13.

[97] M. Ganguli, R. Bhardwaj, A. Sankaranarayanan, S. Raghavan, S. Sesha, G. Hyatt, M. Sundararajan, A. Chylinski, and A. Prakash, "CPU overprovisioning and cloud compute workload scheduling mechanism," 3 2018, uS Patent 9,921,866.

[98] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: Towards automated SLOs for enterprise clusters," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*, 2016, pp. 117–134.

[99] M. D. Marr and M. D. Klein, "Automated profiling of resource usage," 4 2016, uS Patent 9,323,577.

[100] I. Prachitmutita, W. Aittinonmongkol, N. Pojjanasuksakul, M. Supattatham, and P. Padungweang, "Auto-scaling microservices on IaaS under SLA with cost-effective framework," in *Proceedings of 2018 Tenth International Conference on Advanced Computational Intelligence (ICACI 2018)*. IEEE, 2018, pp. 583–588.

[101] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand et al., "Autopilot: workload autoscaling at Google," in *Proceedings of the 15th European Conference on Computer Systems (EuroSys 2020)*, 2020, pp. 1–16.

[102] C. Delimitrou and C. Kozyrakis, "Amdahl's law for tail latency," *Communications of the ACM*, vol. 61, no. 8, pp. 65–72, 2018.

[103] I. Adan and J. Resing, *Queueing theory*. Eindhoven University of Technology Eindhoven, 2002.

[104] J. Mars and L. Tang, "Whare-Map: Heterogeneity in "homogeneous" warehouse-scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 619–630.

[105] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)*, 2020, pp. 733–750.

[106] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro et al., "Applied machine learning at Facebook: A datacenter infrastructure perspective," in *Proceedings of 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*. IEEE, 2018, pp. 620–629.

[107] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload analysis and demand prediction of enterprise data center applications," in *Proceedings of 2007 IEEE 10th International Symposium on Workload Characterization*. IEEE, 2007, pp. 171–180.

[108] "Autoscaling in Kubernetes," https://kubernetes.io/blog/2016/07/autoscaling-in-kubernetes/, Accessed 2020/01/23.

[109] "Instana," https://docs.instana.io/, Accessed 2020/01/23.

[110] "Jaeger: Open source, end-to-end distributed tracing," https://jaegertracing.io/, Accessed 2020/01/23.

[111] "Lightstep distributed tracing," https://lightstep.com/distributed-tracing/, Accessed 2020/01/23.

[112] "SkyWalking: An application performance monitoring system," https://github.com/apache/skywalking, Accessed 2020/01/23.

[113] "OpenZipkin: A distributed tracing system," https://zipkin.io/, Accessed 2020/01/23.

[114] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf

[115] "OpenTracing," https://opentracing.io/, Accessed 2020/01/23.

[116] "Neo4j: Native Graph Database," https://github.com/neo4j/neo4j, Accessed 2020/01/23.

[117] H. Liu, J. Zhang, H. Shan, M. Li, Y. Chen, X. He, and X. Li, "JCallGraph: Tracing microservices in very large scale container cloud platforms," in *Proceedings of International Conference on Cloud Computing*. Springer, 2019, pp. 287–302.

[118] "cAdvisor," https://github.com/google/cadvisor, Accessed 2020/01/23.

[119] "The Prometheus monitoring system and time series database," https://github.com/prometheus/prometheus, Accessed 2020/01/23.

[120] "perf," http://man7.org/linux/man-pages/man1/perf.1.html, Accessed 2020/01/23.

[121] R. H. Lindeman, "Introduction to bivariate and multivariate analysis," Scott Foresman & Co, Tech. Rep., 1980.

[122] S. Tonidandel and J. M. LeBreton, "Relative importance analysis: A useful supplement to regression analysis," *Journal of Business and Psychology*, vol. 26, no. 1, pp. 1–9, 2011.

[123] S. White and P. Smyth, "Algorithms for estimating relative importance in networks," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003, pp. 266–275.

[124] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise Reduction in Speech Processing*. Springer, 2009, pp. 1–4.

[125] R. Eisinga, M. Te Grotenhuis, and B. Pelzer, "The reliability of a two-item scale: Pearson, Cronbach, or Spearman-Brown?" *International Journal of Public Health*, vol. 58, no. 4, pp. 637–642, 2013.

[126] C. P. Diehl and G. Cauwenberghs, "SVM incremental learning, adaptation and optimization," in *Proceedings of 2003 International Joint Conference on Neural Networks*, vol. 4. IEEE, 2003, pp. 2685–2690.

[127] P. Laskov, C. Gehl, S. Krüger, and K.-R. Müller, "Incremental support vector learning: Analysis, implementation and applications," *Journal of Machine Learning Research*, vol. 7, no. Sep, pp. 1909–1936, 2006.

[128] "scikit-learn," https://scikit-learn.org/stable/, Accessed 2020/01/23.

[129] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *CoRR*, vol. abs/1708.05866, 2017. [Online]. Available: http://arxiv.org/abs/1708.05866

[130] "AWS auto scaling documentation," https://docs.aws.amazon.com/autoscaling/index.html, Accessed 2020/01/23.

[131] "Azure autoscale," https://azure.microsoft.com/en-us/features/autoscale/, Accessed 2020/01/23.

[132] "Google cloud load balancing and scaling," https://cloud.google.com/compute/docs/load-balancing-and-autoscaling, Accessed 2020/01/23.

[133] 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," *arXiv preprint arXiv:1509.02971*, 2015.

[134] I. Grondman, L. Busoniu, G. A. Lopes, and R. Babuska, "A survey of actor-critic reinforcement learning: Standard and natural policy gradients," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 6, pp. 1291–1307, 2012.

[135] "NumPy," https://numpy.org/doc/stable/index.html, Accessed 2020/01/23.

[136] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey," *Journal of Machine Learning Research*, pp. 1633–1685, 7 2009.

[137] M. E. Taylor, G. Kuhlmann, and P. Stone, "Autonomous transfer for reinforcement learning." in *Proceedings of 2008 International Conference of Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2008, pp. 283–290.

[138] L. A. Celiberto Jr, J. P. Matsuura, R. L. De Màntaras, and R. A. Bianchi, "Using transfer learning to speed-up reinforcement learning: A case-based approach," in *Proceedings of 2010 Latin American Robotics Symposium and Intelligent Robotics Meeting*. IEEE, 2010, pp. 55–60.

[139] "PyTorch," https://pytorch.org/, Accessed 2020/01/23.

[140] "Intel memory bandwidth allocation," https://github.com/intel/intel-cmt-cat, Accessed 2020/01/23.

[141] "Intel cache allocation technology," https://github.com/intel/intel-cmt-cat, Accessed 2020/01/23.

[142] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, “Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems,” in *Proceedings of 2008 IEEE 14th International Symposium on High Performance Computer Architecture (HPCA 2008)*. IEEE, 2008, pp. 367–378.

[143] N. Rafique, W.-T. Lim, and M. Thottethodi, “Architectural support for operating system-driven CMP cache management,” in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*. Association for Computing Machinery, 2006. [Online]. Available: https://doi.org/10.1145/1152154.1152160 pp. 2—-12.

[144] “HTB - Hierarchical Token Bucket,” https://linux.die.net/man/8/tc-htb, Accessed 2020/01/23.

[145] “wrk2: An HTTP benchmarking tool based mostly on wrk,” https://github.com/giltene/wrk2, Accessed 2020/01/23.

[146] “tc: Traffic Control in the Linux kernel,” https://linux.die.net/man/8/tc, Accessed 2020/01/23.

[147] C. Delimitrou and C. Kozyrakis, “iBench: Quantifying interference for datacenter applications,” in *2013 IEEE International Symposium on Workload Characterization (IISWC 2013)*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 23–33.

[148] “stress-ng,” https://wiki.ubuntu.com/Kernel/Reference/stress-ng, Accessed 2020/01/23.

[149] “pmbw: Parallel Memory Bandwidth Benchmark,” https://panthema.net/2013/pmbw/, Accessed 2020/01/23.

[150] “Sysbench,” https://github.com/akopytov/sysbench, Accessed 2020/01/23.

[151] “Trickle: A lightweight userspace bandwidth shaper,” https://linux.die.net/man/1/trickle, Accessed 2020/01/23.

[152] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, “Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study,” *IEEE Transactions on Software Engineering*, vol. 14, no. 8, pp. 1–1, 2018.

[153] A. Sriraman, A. Dhanotia, and T. F. Wenisch, “SoftSKU: optimizing server architectures for microservice diversity@ scale,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 513–526.

[154] T. Ueda, T. Nakaike, and M. Ohara, “Workload characterization for microservices,” in *Proceedings of 2016 IEEE International Symposium on Workload Characterization (IISWC 2016)*. IEEE, 2016, pp. 1–10.

[155] S. Chen, S. GalOn, C. Delimitrou, S. Manne, and J. F. Martinez, "Workload characterization of interactive cloud services on big and small server platforms," in *Proceedings of 2017 IEEE International Symposium on Workload Characterization (IISWC 2017)*. IEEE, 2017, pp. 125–134.

[156] P. Patros, S. A. MacKay, K. B. Kent, and M. Dawson, "Investigating resource interference and scaling on multitenant PaaS clouds," in *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, 2016, pp. 166–177.

[157] K. Ott and R. Mahapatra, "Hardware performance counters for embedded software anomaly detection," in *Proceedings of 2018 IEEE 16th Intl. Conf. on Dependable, Autonomic and Secure Computing, the 16th Intl. Conf. on Pervasive Intelligence and Computing, the 4th Intl. Conf. on Big Data Intelligence and Computing and Cyber Science and Technology Congress*. IEEE, 2018, pp. 528–535.

[158] X. Wang, S. Chai, M. Isnardi, S. Lim, and R. Karri, "Hardware performance counter-based malware identification and detection with adaptive compressive sensing," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1–23, 2016.

[159] 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)*. Washington, DC, USA: IEEE Computer Society, 2019, pp. 122–132.

[160] R. Chandramouli and Z. Butcher, "Building secure microservices-based applications using service-mesh architecture," *NIST Special Publication*, vol. 800, p. 204A, 2020.

[161] H. Qiu, Y. Chen, T. Xu, Z. T. Kalbarczyk, and R. K. Iyer, "SLO beyond the hardware isolation limits," 2021.

[162] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker, "X-trace: A pervasive network tracing framework," in *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 2007)*, 2007, pp. 271–284.

[163] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, 2014, pp. 217–231.

[164] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi et al., "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 34–50.

[165] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modelling." in *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI 2004)*, vol. 4, 2004, pp. 1–14.

[166] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 595–604.

[167] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: Actionable insights from monitored metrics in distributed systems," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017, pp. 14–27.

[168] "Sysdig," https://sysdig.com/, Accessed 2020/01/23.

[169] S. Y. Shah, X.-H. Dang, and P. Zerfos, "Root cause detection using dynamic dependency graphs from time series data," in *Proceedings of 2018 IEEE International Conference on Big Data (Big Data 2018)*. IEEE, 2018, pp. 1998–2003.

[170] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proceedings of the 26th International Conference on World Wide Web (WWW 2017)*, 2017, pp. 469–478.

[171] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Proceedings of 2018 International Conference on Service-Oriented Computing*. Springer, 2018, pp. 3–20.

[172] J. Weng, J. H. Wang, J. Yang, and Y. Yang, "Root cause analysis of anomalies of multitier services in public clouds," *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, pp. 1646–1659, 2018.

[173] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root cause localization of performance issues in microservices," in *Proceedings of 2020 IEEE/IFIP Network Operations and Management Symposium (NOMS 2020)*, 2020, pp. 1–9.

[174] S. Jha, S. Cui, S. Banerjee, T. Xu, J. Enos, M. Showerman, Z. T. Kalbarczyk, and R. K. Iyer, "Live forensics for HPC systems: A case study on distributed storage systems," in *Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis*, 2020.

[175] G. Jeh and J. Widom, "Scaling personalized web search," in *Proceedings of the 12th International Conference on World Wide Web*, 2003, pp. 271–279.

[176] P. Nguyen and K. Nahrstedt, "Monad: Self-adaptive micro-service infrastructure for heterogeneous scientific workflows," in *Proceedings of the 2017 IEEE International Conference on Autonomic Computing (ICAC 2017)*. IEEE, 2017, pp. 187–196.

[177] P. V. Nguyen, "Dossier: Distributed operating system and infrastructure for scientific data management," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2018.

[178] P. Nguyen and K. Nahrstedt, "Resource management for elastic publish subscribe systems: A performance modeling-based approach," in *Proceedings of the 2016 IEEE 9th International Conference on Cloud Computing (CLOUD 2016).* IEEE, 2016, pp. 561–568.

[179] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2011).* Berkeley, CA, USA: USENIX Association, 2011, pp. 1–14.

[180] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling scheduling speed and quality in large shared clusters," in *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC 2015)*, 2015, pp. 97–110.

[181] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.

[182] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[183] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "DeepDive: Transparently identifying and managing performance interference in virtualized environments," in *Proceedings of 2013 USENIX Annual Technical Conference (USENIX ATC 2013)*, 2013, pp. 219–230.

[184] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-Clouds: Managing performance interference effects for QoS-aware clouds," in *Proceedings of the 5th European Conference on Computer Systems (EuroSys 2010)*, 2010, pp. 237–250.

[185] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy et al., *Serverless Computing: Current Trends and Open Problems*. Singapore: Springer Singapore, 2017, pp. 1–20.

[186] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the USENIX 2020 Annual Technical Conference (ATC 2020).* Berkeley, CA, USA: USENIX Association, 2020, pp. 205–218.

[187] 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 7th International Workshop on Serverless Computing (WoSC7) 2021.* New York, NY, USA: Association for Computing Machinery, 2021, pp. 1–8.

[188] Z. Wen, Y. Wang, and F. Liu, “StepConf: SLO-aware dynamic resource configuration for serverless function workflows,” in *IEEE Conference on Computer Communications (INFOCOM 2022)*. Washington, DC, USA: IEEE Computer Society, 2022, pp. 1–10.

[189] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, “INFless: A native serverless system for low-latency, high-throughput inference,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 768–781.

[190] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, “Atoll: A scalable low-latency serverless platform,” in *Proceedings of the 12th ACM Symposium on Cloud Computing (SoCC 2021)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 138–152.

[191] Z. Jia and E. Witchel, “Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. New York, NY, USA: Association for Computing Machinery, 2021, p. 152–166.

[192] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, “ENSURE: Efficient scheduling and autonomous resource management in serverless environments,” in *International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS 2020)*. Washington, DC, USA: IEEE Computer Society, 2020, pp. 1–10.

[193] M. Mastrolilli and O. Svensson, “(acyclic) job shops are hard to approximate,” in *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 583–592.

[194] 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)*. Cambridge, MA, USA: PMLR, 7 2020, pp. 629–641.

[195] L. Schuler, S. Jamil, and N. Kühl, “AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments,” in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid 2021)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 804–811.

[196] A. Zafeiropoulos, E. Fotopoulou, N. Filinis, and S. Papavassiliou, “Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms,” *Simulation Modelling Practice and Theory*, vol. 116, 2022.

[197] 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)*. Washington, DC, USA: IEEE Computer Society, 2021, pp. 31–40.

[198] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, “Autopilot: Workload autoscaling at Google,” in *Proceedings of the 15th European Conference on Computer Systems (EuroSys 2020)*. New York, NY, USA: Association for Computing Machinery, 2020.

[199] 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)*, vol. 32, no. 3, pp. 514–526, 2020.

[200] Y. Garí, D. A. Monge, E. Pacini, C. Mateos, and C. G. Garino, “Reinforcement learning-based application autoscaling in the cloud: A survey,” *Engineering Applications of Artificial Intelligence*, vol. 102, 2021.

[201] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell, “Hipster: Hybrid task manager for latency-critical cloud workloads,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA 2017)*. Washington, DC, USA: IEEE Computer Society, 2017, pp. 409–420.

[202] T. Veni and M. S. Bhanu, “Auto-scale: Automatic scaling of virtualized resources using neuro-fuzzy reinforcement learning approach,” *International Journal of Big Data Intelligence*, vol. 3, no. 3, pp. 145–153, 2016.

[203] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan et al., “Ray: A distributed framework for emerging AI applications,” in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*. Berkeley, CA, USA: USENIX Association, 2018, pp. 561–577.

[204] L. Buşoniu, R. Babuška, and B. D. Schutter, “Multi-agent reinforcement learning: An overview,” *Innovations in Multi-agent Systems and Applications*, vol. 331, no. 1, pp. 183–221, 2010.

[205] K. Zhang, Z. Yang, and T. Başar, “Multi-agent reinforcement learning: A selective overview of theories and algorithms,” *Handbook of Reinforcement Learning and Control*, vol. 325, pp. 321–384, 2021.

[206] K. Zhang, Z. Yang, and T. Başar, “Decentralized multi-agent reinforcement learning with networked agents: Recent advances,” *Frontiers of Information Technology & Electronic Engineering*, vol. 22, no. 6, pp. 802–814, 2021.

[207] O. Buffet, A. Dutech, and F. Charpillet, “Incremental reinforcement learning for designing multi-agent systems,” in *Proceedings of the 5th International Conference on Autonomous Agents (AGENT 2001)*. New York, NY, USA: Association for Computing Machinery, 2001, p. 31–32.

[208] J. Zhao and J. Schmidhuber, “Incremental self-improvement for life-time multi-agent reinforcement learning,” in *Proceedings of the 4th International Conference on Simulation of Adaptive Behavior*. Cambridge, MA, USA: Citeseer, 1996, pp. 516–525.

[209] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[210] 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 *Proceedings of the 4th International Conference on Learning Representations (ICLR 2016)*. Online: Open Review, 2016. [Online]. Available: https://arxiv.org/abs/1509.02971

[211] H. Van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double Q-learning,” in *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI 2016)*, vol. 30. Palo Alto, CA, USA: AAAI Press, 2016.

[212] M. Tan, “Multi-agent reinforcement learning: Independent vs. cooperative agents,” in *Proceedings of the 10th International Conference on Machine Learning (ICML 1993)*. Cambridge, MA, USA: PMLR, 1993, pp. 330–337.

[213] L. Zheng, J. Yang, H. Cai, M. Zhou, W. Zhang, J. Wang, and Y. Yu, “Magent: A many-agent reinforcement learning platform for artificial collective intelligence,” in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 2018)*, vol. 32. Palo Alto, CA, USA: AAAI Press, 2018.

[214] A. Angiuli, J.-P. Fouque, and M. Lauriere, “Reinforcement learning for mean field games, with applications to economics,” 2021. [Online]. Available: https://arxiv.org/abs/2106.13755

[215] Y. Yang, R. Luo, M. Li, M. Zhou, W. Zhang, and J. Wang, “Mean field multi-agent reinforcement learning,” in *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*. Cambridge, MA, USA: PMLR, 2018, pp. 5571–5580.

[216] N. Saldi, T. Basar, and M. Raginsky, “Markov–Nash equilibria in mean-field games with discounted cost,” *SIAM Journal on Control and Optimization*, vol. 56, no. 6, pp. 4256–4287, 2018.

[217] N. Saldi, T. Başar, and M. Raginsky, “Approximate Nash equilibria in partially observed stochastic games with mean-field interactions,” *Mathematics of Operations Research*, vol. 44, no. 3, pp. 1006–1033, 2019.

[218] W. Mao, H. Qiu, C. Wang, H. Franke, Z. Kalbarczyk, R. Iyer, and T. Başar, "A mean-field game approach to cloud resource management with function approximation," in *Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS 2022)*, vol. 35. Curran Associates, Inc., 2022, pp. 36 243–36 258.

[219] A. S. Foundation, "OpenWhisk," https://github.com/apache/openwhisk, 2022, accessed: 2022-04-14.

[220] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of Function-as-a-Service computing," in *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO 2019)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1063–1075.

[221] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Serverlessbench (socc 2020)," https://github.com/SJTU-IPADS/ServerlessBench, 2020.

[222] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "SeBS: A serverless benchmark suite for Function-as-a-Service computing," in *Proceedings of the 22nd International Middleware Conference (Middleware 2021)*. New York, NY, USA: Association for Computing Machinery, 2021, p. 64–78.

[223] P. Turner, B. B. Rao, and N. Rao, "CPU bandwidth control for CFS," in *Proceedings of the Linux Symposium*. Ottawa, Ontario, Canada: Linux Symposium, 2010, pp. 245–254.

[224] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like clockwork: Performance predictability from the bottom up," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*. Berkeley, CA, USA: USENIX Association, 2020, pp. 443–462.

[225] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2016.

[226] D. Crankshaw, G.-E. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov, "InferLine: Latency-aware provisioning and scaling for prediction serving pipelines," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC 2020)*. New York, NY, USA: Association for Computing Machinery, 2020, p. 477–491.

[227] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "GrandSLAm: Guaranteeing SLAs for jobs in microservices execution frameworks," in *Proceedings of the 14th European Conference on Computer Systems (EuroSys 2019)*. New York, NY, USA: Association for Computing Machinery, 2019.

[228] S. Dev, D. Lo, L. Cheng, and P. Ranganathan, "Autonomous warehouse-scale computers," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–6.

[229] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-scale Machines*, 3rd ed. San Rafael, CA, USA: Morgan & Claypool Publishers, 2018.

[230] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA 2015)*. New York, NY, USA: Association for Computing Machinery, 2015, p. 450–462.

[231] F. Kalim, L. Xu, S. Bathey, R. Meherwal, and I. Gupta, "Henge: Intent-driven multi-tenant stream processing," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC 2018)*. New York, NY, USA: Association for Computing Machinery, 2018, p. 249–262.

[232] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling Quality-of-Service in serverless computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC 2020)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 311–327.

[233] V. Mittal, S. Qi, R. Bhattacharya, X. Lyu, J. Li, S. G. Kulkarni, D. Li, J. Hwang, K. Ramakrishnan, and T. Wood, "Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds," in *Proceedings of the 12th ACM Symposium on Cloud Computing (SoCC 2021)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 168–181.

[234] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Practical scheduling for real-world serverless computing," *arXiv preprint arXiv:2111.07226*, 2021.

[235] 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)*, 2022, pp. 438–448.

[236] Q. Li, B. Li, P. Mercati, R. Illikkal, C. Tai, M. Kishinevsky, and C. Kozyrakis, "RAMBO: Resource allocation for microservices using bayesian optimization," *IEEE Computer Architecture Letters*, vol. 20, pp. 46–49, 2021.

[237] Amazon, "AWS Lambda concurrency limit," https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html, 2022, accessed: 2022-04-14.

[238] C. S. de Witt, T. Gupta, D. Makoviichuk, V. Makoviychuk, P. H. Torr, M. Sun, and S. Whiteson, "Is independent learning all you need in the StarCraft multi-agent challenge?" *arXiv:2011.09533*, vol. abs/2011.09533, 2020.

[239] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3–4, p. 229–256, May 1992.

[240] S. Ginzburg and M. J. Freedman, "Serverless isn't server-less: Measuring and exploiting resource variability on cloud FaaS platforms," in *Proceedings of the 6th International Workshop on Serverless Computing (WoSC6) 2020*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 43–48.

[241] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984.

[242] T. Moors, "A critical review of "end-to-end arguments in system design"," in *Proceedings of IEEE International Conference on Communications 2002 (ICC 2002)*, vol. 2. Washington, DC, USA: IEEE Computer Society, 2002, pp. 1214–1219.

[243] L. S. Shapley, "Stochastic games," *Proceedings of the National Academy of Sciences*, vol. 39, no. 10, pp. 1095–1100, 1953.

[244] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 559–572.

[245] C. Yu, A. Velu, E. Vinitsky, Y. Wang, A. Bayen, and Y. Wu, "The surprising effectiveness of PPO in cooperative, multi-agent games," 2021. [Online]. Available: https://arxiv.org/abs/2103.01955

[246] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya et al., "Hydra: A federated resource manager for data-center scale analytics," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2019)*. Berkeley, CA, USA: USENIX Association, 2019, pp. 177–192.

[247] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC 2013)*. New York, NY, USA: Association for Computing Machinery, 2013.

[248] CNCF, "Kubernetes," https://kubernetes.io/, 2022, accessed: 2022-04-14.

[249] Kubernetes, "Horizontal Pod Autoscaling," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2022, accessed: 2022-04-14.

[250] K. Sripanidkulchai, S. Sahu, Y. Ruan, A. Shaikh, and C. Dorai, "Are clouds ready for large distributed applications?" *SIGOPS Operating Systems Review.*, vol. 44, no. 2, p. 18–23, April 2010.

[251] 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)*, 2019, pp. 329–338.

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

[253] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2017)*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 197–210.

[254] 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)*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 218–235.

[255] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA 2015)*, 2015, pp. 450–462.

[256] S. Dev, D. Lo, L. Cheng, and P. Ranganathan, "Autonomous warehouse-scale computers," in *ACM/IEEE 57th Design Automation Conference (DAC 2020)*, 2020, pp. 1–6.

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

[258] G. Cloud, "Google cloud load balancing and autoscaling," https://cloud.google.com/compute/docs/load-balancing-and-autoscaling, 2022, accessed: 2022-11-23.

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

[260] Azure, "Azure autoscale," https://azure.microsoft.com/en-us/features/autoscale/, 2022, accessed: 2022-11-23.

[261] 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.

[262] 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)*. Red Hook, NY, USA: Curran Associates Inc., 2013, pp. 3111–3119.

[263] 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)*. USA: Association for Computational Linguistics, 2010, pp. 384–394.

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

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

[266] 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*, vol. 38, no. 1, pp. 32–39, 2021.

[267] 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).* New York, NY, USA: Association for Computing Machinery, 2021, pp. 724–739.

[268] OpenAI, "OpenAI Gym documentation," https://www.gymlibrary.dev/, 2022, accessed: 2022-11-23.

[269] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aab3050

[270] 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).* JMLR.org, 2017, pp. 1126–1135.

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

[272] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey, "Meta-learning in neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 09, pp. 5149–5169, 2022.

[273] 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.

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

[275] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[276] 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)*. JMLR.org, 2016, pp. 1842–1850.

[277] S. Fujimoto, D. Meger, and D. Precup, "Off-policy deep reinforcement learning without exploration," in *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*. PMLR, 2019, pp. 2052–2062.

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

[279] G. G. Documentation, "Configuring multidimensional Pod autoscaling in GKE," https://cloud.google.com/kubernetes-engine/docs/how-to/multidimensional-pod-autoscaling, 2022, accessed: 2022-11-23.

[280] InfluxData, "InfluxDB," https://github.com/influxdata/influxdb, 2022, accessed: 2022-11-23.

[281] S. L. Ho and M. Xie, "The use of ARIMA models for reliability forecasting and analysis," *Computers & Industrial Engineering*, vol. 35, no. 1-2, pp. 213–216, 1998.

[282] 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.

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

[284] A. Yang, C. Lu, J. Li, X. Huang, T. Ji, X. Li, and Y. Sheng, "Application of meta-learning in cyberspace security: A survey," *Digital Communications and Networks*, vol. 9, no. 1, pp. 67–78, 2023.

[285] 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)*, 2015, pp. 1521–1526.

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

[287] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*. New York, NY, USA: Association for Computing Machinery, 2019, p. 107–120.

[288] AWS, “Amazon SageMaker,” https://aws.amazon.com/sagemaker/, 2022, accessed: 2022-11-23.

[289] Z. Ding, T. Yu, H. Zhang, Y. Huang, G. Li, Q. Guo, L. Mai, and H. Dong, “Efficient reinforcement learning development with RLzoo,” in *Proceedings of the 29th ACM International Conference on Multimedia (MM 2021)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 3759–3762.

[290] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone, “Curriculum learning for reinforcement learning domains: A framework and survey,” *Journal of Machine Learning Research*, vol. 21, no. 1, 2022, https://jmlr.org/papers/volume21/20-212/20-212.pdf.

[291] M. Panzer and B. Bender, “Deep reinforcement learning in production systems: A systematic literature review,” *International Journal of Production Research*, vol. 60, no. 13, pp. 4316–4341, 2022.

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

[293] S. Xue, C. Qu, X. Shi, C. Liao, S. Zhu, X. Tan, L. Ma, S. Wang, S. Wang, Y. Hu, L. Lei, Y. Zheng, J. Li, and J. Zhang, “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)*. New York, NY, USA: Association for Computing Machinery, 2022, p. 4290–4299.

[294] H. L. Leka, Z. Fengli, A. T. Kenea, N. W. Hundera, T. G. Tohye, and A. T. Tegene, “PSO-based ensemble meta-learning approach for cloud virtual machine resource usage prediction,” *Symmetry*, vol. 15, no. 3, p. 613, 2023.

[295] G. Qu, H. Wu, R. Li, and P. Jiao, “DMRO: A deep meta reinforcement learning-based task offloading framework for edge-cloud computing,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 3448–3459, 2021.

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

[297] Z. Zhang, Q. Liu, S. Qiu, S. Zhou, and C. Zhang, “Unknown attack detection based on zero-shot learning,” *IEEE Access*, vol. 8, pp. 193 981–193 991, 2020.

[298] W. Liang, Y. Hu, X. Zhou, Y. Pan, and K. I.-K. Wang, “Variational few-shot learning for microservice-oriented intrusion detection in distributed industrial IoT,” *IEEE Transactions on Industrial Informatics*, vol. 18, no. 8, pp. 5087–5095, 2022.

[299] V. K. Verma, D. Brahma, and P. Rai, “Meta-learning for generalized zero-shot learning,” in *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 6062–6069.

[300] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone, “Curriculum learning for reinforcement learning domains: A framework and survey,” *Journal of Machine Learning Research*, vol. 21, no. 1, jan 2020.

[301] 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).* Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[302] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, “Google-Wide Profiling: A continuous profiling infrastructure for data centers,” *IEEE Micro*, vol. 30, no. 4, pp. 65–79, 2010.

[303] C. Adams, L. Alonso, B. Atkin, J. Banning, S. Bhola, R. Buskens, M. Chen, X. Chen, Y. Chung, Q. Jia, N. Sakharov, G. Talbot, A. Tart, and N. Taylor, “Monarch: Google’s planet-scale in-memory time series database,” *Proceedings of the VLDB Endowment*, vol. 13, no. 12, p. 3181–3194, 8 2020.

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

[305] H. Qiu, W. Mao, C. W. H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, “On the promise and challenges of foundation models for learning-based cloud systems management,” in *Workshop on Machine Learning for Systems at NeurIPS 2023*, 2023.

[306] S. Lee, Y. Seo, K. Lee, P. Abbeel, and J. Shin, “Offline-to-online reinforcement learning via balanced replay and pessimistic Q-ensemble,” in *Proceedings of the 5th Conference on Robot Learning (CoRL 2021).* PMLR, 2021, pp. 1702–1712.

[307] D. Patterson, J. Gonzalez, U. Hölzle, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. R. So, M. Texier, and J. Dean, “The carbon footprint of machine learning training will plateau, then shrink,” *Computer*, vol. 55, no. 7, pp. 18–28, 2022.

[308] A. S. Luccioni, Y. Jernite, and E. Strubell, “Power hungry processing: Watts driving the cost of AI deployment?” *arXiv preprint arXiv:2311.16863*, 2023.

[309] C. Olston, F. Li, J. Harmsen, J. Soyke, K. Gorovoy, L. Lao, N. Fiedel, S. Ramesh, and V. Rajashekhar, “TensorFlow-Serving: Flexible, high-performance ML serving,” in *Workshop on ML Systems at NIPS 2017*, 2017.

[310] C. Zhang, M. Yu, W. Wang, and F. Yan, “MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving,” in *Proceedings of the USENIX 2019 Annual Technical Conference (ATC 2019).* Berkeley, CA, USA: USENIX Association, 2019, pp. 1049–1062.

[311] H. Zhang, Y. Tang, A. Khandelwal, and I. Stoica, “Shepherd: Serving DNNs in the wild,” in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2023)*, 2023, pp. 787–808.

[312] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez et al., “AlpaServe: Statistical multiplexing with model parallelism for deep learning serving,” in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2023)*, 2023, pp. 663–679.

[313] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, “Clipper: A low-latency online prediction serving system,” in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2017)*, 2017, pp. 613–627.

[314] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, “INFaaS: Automated model-less inference serving,” in *Proceedings of 2021 USENIX Annual Technical Conference (ATC 2021)*, 2021, pp. 397–411.

[315] J. R. Gunasekaran, C. S. Mishra, P. Thinakaran, B. Sharma, M. T. Kandemir, and C. R. Das, “Cocktail: A multidimensional optimization for model serving in cloud,” in *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2022)*, 2022, pp. 1041–1057.

[316] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, “Orca: A distributed serving system for Transformer-based generative models,” in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, 2022, pp. 521–538.

[317] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, “Fast distributed inference serving for large language models,” *arXiv preprint arXiv:2305.05920*, 2023.

[318] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, “Efficient memory management for large language model serving with PagedAttention,” in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 2023)*, 2023, pp. 611–626.

[319] J. Fang, Y. Yu, C. Zhao, and J. Zhou, “TurboTransformers: An efficient GPU serving system for transformer models,” in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2021)*, 2021, pp. 389–402.

[320] F. Wang, W. Zhang, S. Lai, M. Hao, and Z. Wang, “Dynamic GPU energy optimization for machine learning training workloads,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2943–2954, 2021.

[321] J. You, J.-W. Chung, and M. Chowdhury, “Zeus: Understanding and optimizing GPU energy consumption of DNN training,” in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2023)*, 2023, pp. 119–139.

[322] S. Choi, I. Koo, J. Ahn, M. Jeon, and Y. Kwon, “EnvPipe: Performance-preserving DNN training framework for saving energy,” in *Proceedings of the 2023 USENIX Annual Technical Conference (ATC 2023)*, 2023, pp. 851–864.

[323] J.-W. Chung, Y. Gu, I. Jang, L. Meng, N. Bansal, and M. Chowdhury, "Perseus: Removing energy bloat from large model training," *arXiv preprint arXiv:2312.06902*, 2023.

[324] P. Patel, Z. Gong, S. Rizvi, E. Choukse, P. Misra, T. Anderson, and A. Sriraman, "Towards improved power management in cloud GPUs," *IEEE Computer Architecture Letters*, vol. 22, no. 2, pp. 141–144, 2023.

[325] G. X. Team, "XLA: Optimizing compiler for machine learning," 2017.

[326] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016)*, June 2016.

[327] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional Transformers for language understanding," in *Proceedings of the 2019 North American Chapter of the Association for Computational Linguistics (ACL 2019)*, 2019. [Online]. Available: https://arxiv.org/pdf/1810.04805.pdf

[328] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz et al., "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020)*, 2020, pp. 38–45.

[329] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters," in *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2022)*, 2022, pp. 945–960.

[330] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing et al., "Alpa: Automating inter-and intra-operator parallelism for distributed deep learning," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, 2022, pp. 559–578.

[331] L. Zheng, W.-L. Chiang, Y. Sheng, T. Li, S. Zhuang, Z. Wu, Y. Zhuang, Z. Li, Z. Lin, E. Xing et al., "LMSYS-Chat-1M: A large-scale real-world LLM conversation dataset," *arXiv preprint arXiv:2309.11998*, 2023. [Online]. Available: https://arxiv.org/pdf/2309.11998.pdf

[332] Y. Xu, H. Lee, D. Chen, B. Hechtman, Y. Huang, R. Joshi, M. Krikun, D. Lepikhin, A. Ly, M. Maggioni et al., "GSPMD: General and scalable parallelization for ML computation graphs," *arXiv preprint arXiv:2105.04663*, 2021.

[333] W. Zhang, W. Wei, L. Xu, L. Jin, and C. Li, "AI-Matrix: A deep learning benchmark for Alibaba data centers," *arXiv preprint arXiv:1909.10562*, 2019.

[334] K. Jansen, S. Kratsch, D. Marx, and I. Schlotter, "Bin packing with fixed number of bins revisited," *Journal of Computer and System Sciences*, vol. 79, no. 1, pp. 39–49, 2013.

[335] C. Winship and R. D. Mare, "Regression models with ordinal variables," *American Sociological Review*, pp. 512–525, 1984.

[336] F. Bang and F. Di, "GPTCache: An open-source semantic cache for LLM applications enabling faster answers and cost savings," in *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, 2023, pp. 212–218.

[337] B. Zurita Ares, P. G. Park, C. Fischione, A. Speranzon, and K. H. Johansson, "On power control for wireless sensor networks: System model, middleware component and experimental evaluation," in *2007 European Control Conference (ECC 2007)*, 2007, pp. 4293–4300.

[338] Y. Shadmi, "Fluid limits for shortest job first with aging," *Queueing Systems*, vol. 101, no. 1-2, pp. 93–112, 2022.

[339] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "PipeSwitch: Fast pipelined context switching for deep learning applications," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, 2020, pp. 499–514.

[340] Y. Zhang, Í. 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)*, 2021, pp. 724–739.

[341] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, "Characterization and prediction of deep learning workloads in large-scale GPU datacenters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2021)*. New York, NY, USA: Association for Computing Machinery, 2021.

[342] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2009)*. IEEE, 2009, pp. 248–255.

[343] Y. Song, Z. Mi, H. Xie, and H. Chen, "PowerInfer: Fast large language model serving with a consumer-grade GPU," *arXiv preprint arXiv:2312.12456*, 2023, https://arxiv.org/abs/2312.12456.

[344] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "FlexGen: High-throughput generative inference of large language models with a single GPU," in *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*. PMLR, 2023, pp. 31 094–31 116.

[345] Z. Liu, J. Wang, T. Dao, T. Zhou, B. Yuan, Z. Song, A. Shrivastava, C. Zhang, Y. Tian, C. Re et al., "DejaVu: Contextual sparsity for efficient LLMs at inference time," in *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*. PMLR, 2023, pp. 22 137–22 176.

[346] Y. Leviathan, M. Kalman, and Y. Matias, "Fast inference from transformers via speculative decoding," in *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*. PMLR, 2023, pp. 19 274–19 286.

[347] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, Z. Wang, R. Y. Y. Wong, Z. Chen, D. Arfeen, R. Abhyankar, and Z. Jia, "SpecInfer: Accelerating generative LLM serving with speculative inference and token tree verification," *arXiv preprint arXiv:2305.09781*, 2023, https://arxiv.org/abs/2305.09781.

[348] T. Cai, Y. Li, Z. Geng, H. Peng, and T. Dao, "Medusa: Simple framework for accelerating LLM generation with multiple decoding heads," 2023, https://github.com/FasterDecoding/Medusa.

[349] Y. Fu, P. Bailis, I. Stoica, and H. Zhang, "Break the sequential dependency of LLM inference using lookahead decoding," 2023, https://lmsys.org/blog/2023-11-21-lookahead-decoding/.

[350] F. Song and J. Dongarra, "A scalable framework for heterogeneous GPU-based clusters," in *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2012)*, 2012, pp. 91–100.

[351] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, 2020, pp. 463–479.

[352] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, "Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning," in *Proceedings of the 15th European Conference on Computer Systems (EuroSys 2020)*, 2020, pp. 1–16.

[353] J. N. Foerster, Y. M. Assael, N. de Freitas, and S. Whiteson, "Learning to communicate to solve riddles with deep distributed recurrent Q-networks," *arXiv preprint arXiv:1602.02672*, 2016.

[354] I. Ilahi, M. Usama, J. Qadir, M. U. Janjua, A. Al-Fuqaha, D. T. Hoang, and D. Niyato, "Challenges and countermeasures for adversarial attacks on deep reinforcement learning," *IEEE Transactions on Artificial Intelligence*, vol. 3, no. 2, pp. 90–109, 2021.

[355] L. Qiao, W. Wang, and B. Lin, "Learning accurate and interpretable decision rule sets from neural networks," in *Proceedings of the 2021 AAAI Conference on Artificial Intelligence*, vol. 35, no. 5, 2021, pp. 4303–4311.

[356] Y. Dhebar, K. Deb, S. Nageshrao, L. Zhu, and D. Filev, "Toward interpretable-AI policies using evolutionary nonlinear decision trees for discrete-action systems," *IEEE Transactions on Cybernetics*, 2022.

[357] Z. Wang, W. Zhang, N. Liu, and J. Wang, "Scalable rule-based representation learning for interpretable classification," *Advances in Neural Information Processing Systems*, vol. 34, pp. 30 479–30 491, 2021.

[358] A. Dethise, M. Canini, and N. Narodytska, "Analyzing learning-based networked systems with formal verification," in *Proceedings of the 2021 IEEE Conference on Computer Communications (INFOCOMM 2021)*. IEEE, 2021, pp. 1–10.

[359] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira, "Verifying learning-augmented systems," in *Proceedings of the 2021 ACM Annual Conference of the Special Interest Group on Data Communication (SIGCOMM 2021)*, 2021, pp. 305–318.

[360] K. W. Church, Z. Chen, and Y. Ma, "Emerging trends: A gentle introduction to fine-tuning," *Natural Language Engineering*, vol. 27, no. 6, pp. 763–778, 2021.

[361] 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*, vol. 53, no. 3, jun 2020. [Online]. Available: https://doi.org/10.1145/3386252

[362] 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)*, 2023, pp. 15 211–15 222.

[363] H. Qiu, W. Mao, A. Patke, S. Cui, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "FLASH: Fast model adaptation in ML-centric cloud platforms," in *Proceedings of the 7th Annual Conference on Machine Learning and Systems (MLSys 2024)*, 2024.

[364] W. Mao, H. Qiu, C. Wang, H. Franke, Z. Kalbarczyk, R. Iyer, and T. Başar, "Multi-agent meta-reinforcement learning: Sharper convergence rates with task similarity," in *Proceedings of the 37th Annual Conference on Neural Information Processing Systems (NeurIPS 2023)*, vol. 36. Curran Associates, Inc., 2023, pp. 66 556–66 570.

[365] R. K. Iyer, Z. T. Kalbarczyk, and N. M. Nakka, *Dependable Computing: Design and Assessment*. John Wiley & Sons, 2024.

[366] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, "Detecting silent data corruptions in the wild," 2022.

[367] Y. Du, T. Mukherjee, and T. DebRoy, "Physics-informed machine learning and mechanistic modeling of additive manufacturing to reduce defects," *Applied Materials Today*, vol. 24, p. 101123, 2021.

[368] Z. Cao, X. Zhou, H. Hu, Z. Wang, and Y. Wen, "Toward a systematic survey for carbon neutral data centers," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 2, pp. 895–936, 2022.

[369] P. P. Shinde and S. Shah, "A review of machine learning and deep learning applications," in *Proceedings of the 2018 International Conference on Computing Communication Control and Automation (ICCUBEA 2018).* IEEE, 2018, pp. 1–6.

[370] 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.

[371] J. Gao, H. Wang, and H. Shen, "Smartly handling renewable energy instability in supporting a cloud datacenter," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2020).* IEEE, 2020, pp. 769–778.

[372] A. Radovanović, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care et al., "Carbon-aware computing for datacenters," *IEEE Transactions on Power Systems*, vol. 38, no. 2, pp. 1270–1280, 2022.

[373] R. Arora, U. Devi, T. Eilam, A. Goyal, C. Narayanaswami, and P. Parida, "Towards carbon footprint management in hybrid multicloud," in *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, 2023, pp. 1–7.

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

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

[376] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, p. 1735–1780, nov 1997.

[377] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova, "Well-read students learn better: On the importance of pre-training compact models," *arXiv preprint arXiv:1908.08962*, 2019.