# Trident: A Provider-Oriented Resource Management Framework for Serverless Computing Platforms

Botao Zhu, Yifei Zhu, *Member, IEEE*, Chen Chen, *Member, IEEE*, and Linghe Kong, *Senior Member, IEEE*

***Abstract*—Serverless computing has become increasingly popular due to its flexible and hassle-free service, relieving users from traditional resource management burdens. However, the shift in responsibility has led to unprecedented challenges for serverless providers in managing virtual machines (VMs) and serving heterogeneous function instances. Serverless providers need to purchase, provision and manage VM instances from IaaS providers, aiming to minimize VM provisioning costs while ensuring compliance with Service Level Objectives (SLOs). In this paper, we propose Trident, a provider-oriented resource management framework for serverless computing platforms. Trident optimizes three major serverless computing provisioning problems for serverless providers: workload prediction, VM provisioning, and function placement. Specifically, Trident introduces a novel dynamic model selection algorithm for more accurate workload prediction. With the prediction results, Trident then carefully designs a hierarchical reinforcement learning (HRL)-based approach for VM provisioning with a mix of types and configurations. To further improve resource utilization, Trident employs an effective collocation placement strategy for efficient function container scheduling. Evaluations on the Azure Function dataset demonstrate that Trident maintains the lowest probability of violating SLOs while simultaneously achieving substantial cost savings of up to 71.8% in provisioning expense compared to state-of-the-art methods from industry and academia.**

***Index Terms*—Serverless computing, VM provisioning, hierarchical reinforcement learning, workload prediction.**

## I. Introduction

SERVERLESS computing emerges as a new cloud computing paradigm that provides users with scalable and flexible cloud service [1], [2]. In serverless computing, developers can decompose monolithic applications into fine-grained functions with simplified logic, paying only for function invocations. Serverless providers then pack users' codes into runtime instances (e.g., containers) for request handling and task execution. Serverless computing has been extensively offered by major cloud providers including AWS [3] and Azure [4], as well as the open-source community, such as Apache OpenWhisk [5] and Fission [6].

The paradigm shift from traditional Infrastructure-as-a-Service (IaaS) [7] to serverless computing has transferred the responsibility of resource management entirely to serverless providers. Different from traditional cloud providers, serverless providers may not possess rack servers or enough rack servers to provide serverless computing [8], [9]. It's a common practice for them to acquire the necessary computational resources (i.e., VMs) from internal IaaS departments or external IaaS providers, in a rental model [9]. These resources are provisioned and managed based on the serverless functions' requirements and the promised Service Level Objectives (SLOs) to customers. The scale, type, and duration of these resource usage determine the incurred expenses, which have to be covered by the serverless providers.

To reduce costs and guarantee the promised SLOs, serverless providers should achieve three goals in resource management [10]. First, accurate workload prediction is beneficial [11]. Although it is not essentially a necessity, it might help in provisioning just the right amount of resources at the right time. Second, cost-effective and SLO-guaranteed VM provisioning is essential [12]. Serverless providers must select appropriate VMs and provision the correct number to reduce provisioning costs while meeting the promised SLO. Third, resource-efficient function placement is crucial [13]. Serverless providers must place function containers in provisioned VM instances in a manner that fully utilizes the available resources.

Although previous literature has studied these three aspects, they do not fully achieve the three primary goals in serverless computing. First, existing resource management systems typically rely on a single off-the-shelf prediction model [14], [15], [16] for workload prediction, leading to inaccurate prediction results as we identified. It will cause either high-cost overprovisioning or SLO-violated underprovisionng. Second, their provisioning methods fail to handle the dynamics and complexity of mixed VM provisioning with different types[1] and configurations.[2] They either consider a simplified provisioning scenario with fixed VMs [15], [17], unable to explore cost-saving provisioning policies, or involve high-complexity heuristics [18] or solver-based optimization [14] with poor scalability and performance. Third, existing frameworks pay little attention to the placement of function containers and resort to naive placement heuristics, which can cause severe resource fragmentation and incur additional provisioning costs.

Received 23 January 2025; revised 2 August 2025; accepted 18 August 2025. Date of publication 28 August 2025; date of current version 9 October 2025. This work was supported by the National Natural Science Foundation of China under Grant 62302292. *(Corresponding authors: Yifei Zhu; Chen Chen.)*

Botao Zhu and Yifei Zhu are with the Global College, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: zhubotao@sjtu.edu.cn; yifei.zhu@sjtu.edu.cn).

Chen Chen is with the John Hopcroft Center for Computer Science, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: chen-chen@sjtu.edu.cn).

Linghe Kong is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: linghe.kong@sjtu.edu.cn).

Digital Object Identifier 10.1109/TSC.2025.3603867

[1]VM type refers to the classification of the VM such as spot or on-demand.
[2]Configuration refers to the number of CPU cores and memory of the VM.

In this paper, we propose Trident, a provider-oriented resource management framework for serverless computing platforms. We conduct a thorough data analysis to distill several key insights, and design three modules correspondingly to optimize serverless resource management. First, we find that the selection of the best model for serverless workload prediction exhibits short-term consistency. Additionally, workload sequences, e.g., memory consumption over a certain period, that favor the same model tend to have higher pattern similarity measured by dynamic time warping (DTW) distances. Based on these insights, we introduce a novel model selection module that dynamically selects the best model for each workload sequence. It features a novel triplet buffer to store recent and valuable model selection results, and leverages DTW distance between workload sequences for efficient model selection.

Second, we identify an opportunity for reliable and cost-effective VM provisioning through a combination of spot and on-demand VMs. However, the dynamic nature of spot VMs and serverless functions, together with the complexity of VM combinations, make such mixed VM provisioning rather challenging. We observe a significant frequency gap between the dynamics of spot VMs and those of serverless functions, which enables an elegant decomposition of the provisioning problem. Inspired by this, we incorporate a VM provisioning module to select a mix of VM types and configurations intelligently. A novel hierarchical reinforcement learning (HRL)-based algorithm is proposed to jointly solve the VM selection problem and VM number scaling problem.

Third, we find that by collocating functions with complementary resource demand, the resource utilization can be increased to save cost. To this end, we design a simple yet effective greedy approach for function placement. It jointly considers the resource demand of serverless functions and the capacity of provisioned VMs, collocating resource-complementary functions into VMs with similar CPU-to-memory ratios. It helps mitigate resource fragmentation and avoid unnecessary provisioning costs.

The main contributions of this paper are summarized as follows:

- We conduct extensive data analysis and distill insights about workload prediction and mixed VM provisioning for serverless functions. We identify the shortcomings of existing resource management systems in handling serverless workloads.
- Given these limitations, we design and implement Trident, a provider-oriented system that optimizes end-to-end resource management for serverless computing. It features a dynamic model selection algorithm for more accurate workload prediction, an HRL-based approach for mixed VM provisioning, and an effective function collocation placement algorithm for function placement.
- Experiments on real-world datasets show that our system reduces the cost by up to 71.8% while achieving the lowest SLO violations probability, under 3%, compared to state-of-the-art industrial and academic solutions.

The rest of the paper is structured as follows: Section II introduces the background of serverless computing and the existing resource management framework. Section III presents data analysis, motivation and insights. Section IV describes the detailed design of Trident. Section V presents the implementation and evaluation results of our system on a real-world serverless and VM dataset. Section VI introduces the related work. Section VII concludes the whole paper.

## II. Background and Existing Work

### A. Background

*1) Serverless Computing:* Serverless computing is a cloud model that abstracts infrastructure management, letting developers focus on code while providers handle provisioning, scaling, and maintenance, billing only for actual execution time [9]. Serverless computing enhances flexibility, cost-efficiency, and scalability, making it an appealing solution for event-driven applications, microservices architectures, and dynamic workloads. This model has gained significant adoption in recent years [3], [4], [19] due to its simplicity, rapid deployment capabilities, and the ability to handle fluctuating workloads efficiently. A wide range of applications, such as video processing [20], machine learning [21], and algebraic operations [22], have been taking advantage of serverless computing to improve performance and reduce cost.

*2) Resource Management: A Serverless Provider's View:* In serverless computing, satisfying the promised SLOs and reducing cost are major concerns for serverless providers. To serve customers' functions, they may purchase computing resources from internal IaaS departments or external IaaS providers, managing and optimizing these resources to ensure both cost efficiency and performance reliability. For example, in Microsoft Azure, Azure Functions purchases VM instances from the internal IaaS department of Azure [9], including regular VMs or surplus (e.g., Spot or Burstable) VMs, depending on availability and cost.

Serverless providers need to achieve three goals in resource management to satisfy the promised SLOs and reduce the provisioning costs [10]. First, accurate workload prediction is required [11] [10]. This involves forecasting the future volume of requests or resource demands. An accurate prediction allows serverless providers to better understand the load and effectively determine the number of resources (i.e., VMs) needed to handle incoming traffic, preventing both under-provisioning and over-provisioning, which can lead to performance degradation or unnecessary costs.

Second, cost-effective and SLO-guaranteed VM provisioning is essential [12]. With an accurate workload forecast, providers need to ensure that VM provisioning not only meets the performance requirements defined by the SLOs but also minimizes costs. This can involve strategies such as selecting the right instance types, optimizing resource allocation, and utilizing auto-scaling mechanisms to match resource supply with demand dynamically.

Third, resource-efficient function placement is crucial for optimizing the use of available resources [13]. Once the required number of VMs is determined, it is essential to efficiently place the functions across the available instances, considering factors like load balancing and resource utilization. By optimizing the function placement, providers can further reduce costs while maintaining high performance.

As resource management directly influences both the cost and performance of serverless computing, it is crucial for serverless providers to build a cost-effective and SLO-compliant resource management framework that achieves all of the three goals.

### B. Existing Resource Management Frameworks

Resource management for cloud workloads has been extensively studied in previous literature. AWS officially provides

Reactive Scaling [17] and Prediction Scaling [16] for cloud workloads. The former scales resources based on predefined thresholds and real-time metric monitoring. The latter uses machine learning models (e.g., DeepAR [23]) to predict workloads for resource scaling. TData [24] uses ARIMA and TBATS [25] for flow prediction and autoscaling of Flink jobs. SpotWeb [14] applies cubic spline regression for workload prediction and optimizes VM purchasing based on multi-period portfolio theory. A few works leverage reinforcement learning (RL) to learn policies from the dynamic cloud environment [12], [26]. FIRM [26] uses Deep Deterministic Policy Gradient (DDPG) to allocate resources for microservices. Snape [12] uses constrained RL and dynamically mixes on-demand and spot VMs of the same configuration.

Unfortunately, we find that existing systems fail to achieve the three major goals of serverless resource management: accurate workload prediction, cost-effective and SLO-guaranteed VM provisioning, and resource-efficient function placement. Their limitations will be identified and discussed in the next section through thorough data analysis.

## III. Data Analysis, Motivation and Insight

In this section, we conduct thorough data analysis on existing datasets. We reveal issues of existing resource management systems and distill key insights for the three critical parts of serverless resource management, namely workload prediction, VM provisioning, and function placement. We use the Azure Function dataset [10] to perform workload prediction and analyze the running characteristics of serverless functions. It records the serverless workloads at minute granularity of more than 40,000 functions for two weeks in July 2019. We analyze the characteristics of one popular VM type, spot VM, by investigating its price and eviction rate using the Spot VM dataset [14]. It records the price and eviction rate changes of 58 VMs for nearly 3 months.

### A. Workload Prediction

We investigate the performance of four representative workload prediction models, including two statistical models, ARIMA and TBATS, a machine learning model, Support Vector Regression (SVR) [27], an LSTM-based model, DeepAR (DAR) [23], and an ensemble method, XGBoost (XGB) [28]. They are either representative models for time series prediction or deployed in industrial cloud platforms. From a serverless provider's perspective, we focus on overall resource demand rather than per-function request counts [29], [30] for two reasons: predicting requests for many functions incurs high overhead, and aggregated workload prediction is more robust and accurate [31]. Consequently, we aggregate all functions' workloads using the "AverageAllocatedMb" field of the Azure Function dataset and obtain the overall memory usage. We segment the entire timeline into prediction points at three-minute intervals, a common VM startup time for mainstream IaaS providers [32]. At each prediction point, we are given a 30-minute historical workload sequence to predict future memory usage for 15 minutes. As serverless providers periodically provision resources according to the workload prediction for the next period [16], we analyze fine-grained prediction performance at each prediction point, namely sequence-level prediction. Our main discoveries are summarized as follows:

*Issue. Relying on a single model causes inaccurate workload prediction:* Existing resource management systems typically rely on a single off-the-shelf prediction model [15]–[17], leading to inaccurate prediction results on predicting the everchanging serverless workloads. Fig. 1(a) shows the normalized mean square error (MSE) at sequence level prediction of four models. We normalize the MSE to [0,1] for comparison purposes. We find that the best model at different prediction points varies. There is no "golden model" that is always the best. In Fig. 1(b), we compare the overall prediction performance on the whole dataset between four models and an "oracle" method that selects the best prediction (i.e., prediction with the lowest MSE) from four models at each prediction point. We can see that the prediction error of the worst single model (SVR) is about 3 times of the oracle method, and 1.5 times even for the best single model (ARIMA).

*Insight 1. The selection of the best model shows short-term consistency:* Although the best model changes over time, we find that it remains unchanged over certain periods, exhibiting short-term consistency. For example, at the first 20 points, two statistical models perform much better than the other two models. Then, SVR achieves a lower prediction error from the $25_{th}$ to $45_{th}$ prediction point. Afterward, DeepAR dominates from the $50_{th}$ to $80_{th}$ prediction point.

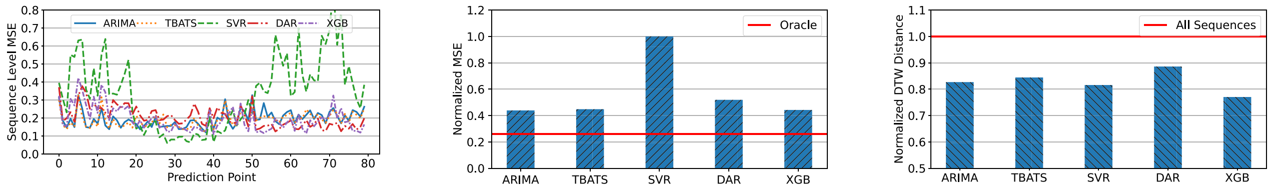
*Insight 2. Workload sequences that favor the same model tend to have higher similarity measured by Dynamic Time Warping (DTW) distances:* We also find that simply relying on short-term consistency for model selection is not enough. There are situations where the selection of the best model shows little short-term consistency. For example, the best model varies from the $45_{th}$ to the $50_{th}$ and from the $20_{th}$ to the $25_{th}$ prediction point. We notice that in addition to the temporal relation between the prediction points, another valuable piece of information is the similarity in workload sequences. To further obtain insights about model selection, we group the history workload sequences by their corresponding best model. We calculate the average of DTW distance [33] between each pair of workload sequences within group $g$ using the following formulation:

$$DTW^g_{avg} = \frac{\sum_{1\leq i<j\leq G} DTW(y_i, y_j)}{G(G-1)}, \tag{1}$$

where $G$ is the size of group $g$, $y_i$ and $y_j$ are two different workload sequences in $g$. The DTW distance is a powerful metric to measure the similarity between two time series. Compared to the simple Euclidean distance, DTW distance is more suitable for time series as it can consider both temporal alignment and shape variations, and is more robust to noise. Formally, the DTW distance between two sequences $y_1$ and $y_2$ with length $K$ is defined as follows :

$$DTW(y_1, y_2) = \min_{W} \sum_{k=1}^{K} \delta(w_k), \tag{2}$$

where $\delta$ is a distance measure between two elements (e.g., Euclidean distance) and $W = [w_1, \ldots, w_p]$ is a warping path that aligns the elements of $y_1$ and $y_2$. We also calculate the average of DTW distances between each pair among all sequences (the red line in Fig. 1(c)), and normalize all the distances by it for comparison purposes. As shown in Fig. 1(c), the sequences within the same best model group have a smaller average DTW distance than that of the whole sequence set. It indicates that

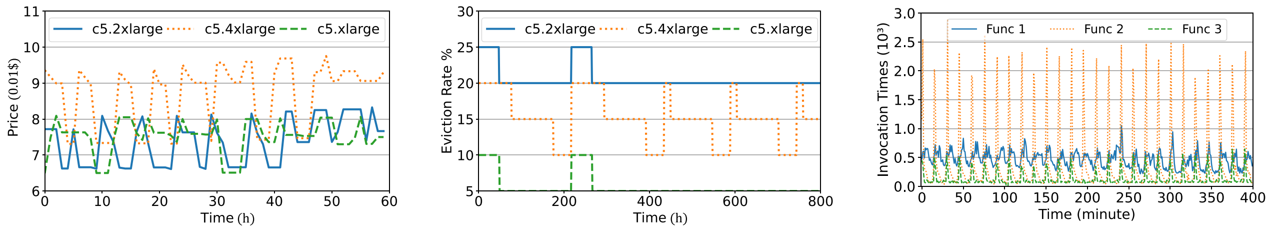


(a) Normalized sequence level MSE of each method at different prediction points.

(b) Normalized MSE of each prediction method on the whole dataset.

(c) Normalized average DTW distance of the sequences that favor the same model.

Fig. 1. The performance of different workload prediction models (ARIMA, TBATS, SVR, DAR and XGB) on Azure Function dataset. "Oracle" is the method that can select the best model at each prediction point. "All Sequences" refers to the average DTW distances among all workload sequences.



(a) Unit price of c5 spot VMs over time.

(b) Eviction rate of c5 spot VMs over time.

(c) Number of function invocations over time.

Fig. 2. Spot VM's running characteristics and serverless function invocations.

TABLE I
DETAILS ABOUT THREE VMS IN FIG. 2

| VM Name | VM Type | VM Configuration | |
|---|---|---|---|
| | | CPU Cores | Memory (GB) |
| c5.xlarge | spot | 4 | 8 |
| c5.2xlarge | spot | 8 | 16 |
| c5.4xlarge | spot | 16 | 32 |

different models may be better at handling workload sequences of specific patterns. This makes DTW an appropriate choice to find the best prediction model.

## B. VM Provisioning

We introduce and investigate spot VM, a popular type of VM considered a promising choice for serverless computing [9]. It is offered at a 70-90% lower price than on-demand VM but may suffer from eviction [14]. Before the eviction, the spot instance will receive an eviction message and is given a grace period (typically 30 seconds) to terminate the running workload. Our main discoveries are summarized as follows:

*Insight 1. Mixed VM provisioning is necessary for cost-effective and SLO-guaranteed serverless computing:* Fig. 2(a) and (b) show the unit price and eviction rate of three c5 spot VMs from Amazon Cloud. The unit price is computed by dividing the VM's actual price by its resource count. The detailed information about these VMs is listed in Table I. As we can infer from these two figures, there is no such VM of the c5 configuration family that is always the cheapest or most reliable over time. Note that the VMs of other families share the same discovery above. The dynamic properties of spot VM require us to adjust the selection of VMs, specified by their type and configuration combinations, in a timely manner for cost-saving and SLO-guaranteed VM provisioning. Further more, to mitigate the unreliability and potential SLO violations caused by the eviction, expensive on-demand VMs are still indispensable for serverless providers. Consequently, mixed VM provisioning with both is necessary to achieve cost-effective and SLO-guaranteed serverless computing.

*Issue. Existing methods fail to handle the dynamics and complexity of mixed VM provisioning:* Most existing resource management frameworks use simple provisioning strategies that adjust the number of VMs with fixed types and configurations [12], [17], [26]. These straightforward approaches often result in high costs and resource wastage due to the varying resource requirements of different functions. Although several studies [14], [18] have considered incorporating heterogeneous VMs, their methods involve high-complexity heuristics [18] or solver-based optimization [14] with poor scalability and performance. Consider a typical VM provisioning scenario in which the number of $n$ distinct VMs is scaled at each time step. The search space for VM provisioning is $m^n$, where $m$ represents the possible VM scaling actions (e.g., increasing or decreasing the number of VMs by a certain percentage [15]). As current IaaS providers offer tens or even hundreds of distinct VMs [34], the search space can become too large to be managed by existing frameworks.

*Insight 2. Frequency gap between the dynamics of functions and spot VM helps decompose the problem:*

We next investigate the function invocation patterns in the Azure Function dataset and the dynamics of spot VM jointly. Comparing Fig. 2(a), (b) and (c), we can see that while the price and eviction rate of spot VM varies by hours [34], serverless workloads fluctuate at a more granular level, often changing in minutes or even mere seconds [4]. Since the price and eviction rate of a VM remains stable for relatively a long period, a selection of VMs can be well-suited or even optimal in this

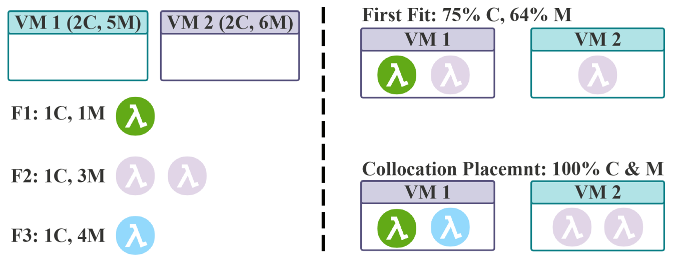


Fig. 3. A simple example of function placement with two different strategies, namely First-Fit and Collocation Placement. "C" is short for CPU cores, "M" is short for GBs of memory.

period and we only need to make scaling decisions on the selected VMs to serve serverless invocations. Assume we select $v$ VMs for a relatively long period (e.g., 1 h). As we fix the selection of VMs, the search space of the original problem at each time step can be reduced to $m^v$ because we only need to adjust the number of $v$ VMs. Though preserving the same order as the previous case, such a reduction could be huge in practice if $v$ is much smaller than $n$.

### C. Function Placement

We discuss the significance of resource-efficient function placement for serverless computing using a simple illustrative example. Our main discoveries are summarized as follows:

*Issue. Ignoring function placement leads to low resource utilization, causing unnecessary cost or SLO violations:* Most existing resource management frameworks resorts to naive placement strategies [12], [14], [26]. We use a simple illustrative example to show the inefficiency of such a naive placement strategy, as presented in Fig. 3. Assume we have two VMs, one with 2 CPU cores and 5 GB of memory, the other with 2 CPU cores and 6 GB of memory. Note that, in practice, all VM instances could have reserved enough resource headroom to avoid potential performance degradation. The serverless workload consists of three functions, namely F1, F2 and F3, with their resource demands placed at the bottom left of the figure. A commonly-used First-Fit strategy would place the function in a sequential order (e.g., F1, F2, and then F3) onto the VM instance that is first found to have enough resources. With that, we can only achieve 75% CPU utilization and 64% memory utilization. Although the total amount of resources is equal with the demand, the workload of F3 can not be served due to resource fragmentation. Under this situation, the serverless providers either need to provision resources with extra cost, or compromise with SLO violations.

*Insight. Collocating resource-complementary functions can increase resource utilization:* The naive First-Fit approach fails to efficiently utilize the resource because it's agnostic to the heterogeneous resource demand of serverless functions and the resource capacity of VMs. Thus, we introduce a new collocation placement method that jointly considers the functions and provisioned VMs, as presented at the bottom right of Fig. 3. We identify that F1 and F3 are resource-complementary in terms of VM 1, since collocating them achieves an identical CPU-to-memory ratio (i.e., 2 / 5 = 0.4) with VM 1. Consequently, we place F1 and F3 in VM 1, and F2 in VM 2. As a result, we increase the resource utilization of both CPU and memory to 100%, and serve all the serverless workloads without introducing extra provisioning costs or causing SLO violations.

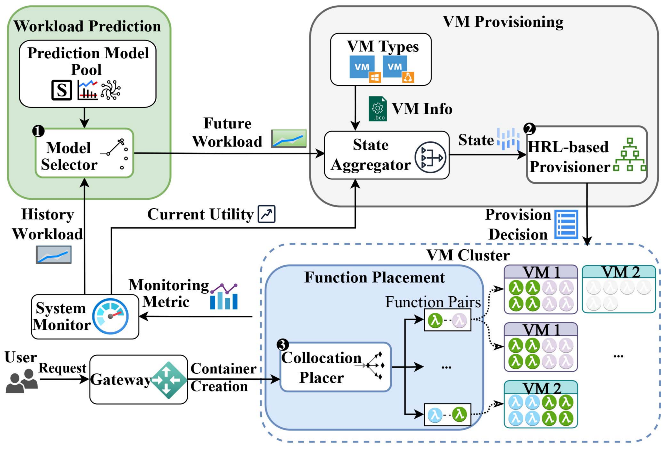


Fig. 4. System overview of Trident.

TABLE II
IMPORTANT NOTATIONS AND SYMBOLS

| Symbol | Description |
|---|---|
| $N$ | Number of functions. |
| $M$ | Number of VMs. |
| $B$ | Buffer used to store triplets for model selection. |
| $L$ | Length of the buffer. |
| $f$ | A prediction model. |
| $e$ | A workload sequence. |
| $e'$ | Ground truth of prediction. |
| $d'$ | DTW distance threshold for model selection. |
| $s_t$ | Achieved SLO at time $t$. |
| $c_t$ | Provisioning costs at time $t$. |
| $x_m^t$ | Number of VM $m$ provisioned at time $t$. |
| $p_m^t$ | Price of VM $m$ at time $t$. |
| $g_n^t$ | Number of function $n$'s invocations served timely at $t$. |
| $q_n^t$ | Number of invocations arrives for function $n$ at time $t$. |
| $u_t$ | Achieved utility at time $t$. |
| $S$ | Promised SLOs by serverless provider. |
| $a_{ij}$ | CPU usage of the function pair $(i,\ j)$. |
| $b_{ij}$ | Memory usage of the function pair $(i,\ j)$. |
| $h_{ij}$ | Maximum number of function pair $(i,\ j)$ for collocation. |
| $r_d$ | Dominant collocation resource ratio for functions. |

## IV. THE TRIDENT FRAMEWORK

### A. System Overview

Fig. 4 provides an overview of our proposed framework, Trident. It consists of three main components: (1) Workload prediction module, which periodically takes the historical workload sequence from the system monitor and selects the best model for predicting future workloads. (2) VM provisioning module, which takes a vectorized system state from the state aggregator and uses an HRL-based provisioner to generate VM provisioning decisions. (3) Function placement module, which runs a collocation placer to collocate the function container into pairs for higher resource utilization and place them on the provisioned VMs. Users can send function invocations through the gateway, which triggers the container creation process. Important notations and symbols are presented in Table II.

### *B. Dynamic Model Selection for Serverless Workload Prediction*

As we've identified in Section III, the best model for serverless workload prediction varies over time. To improve the prediction accuracy, we propose a dynamic model selection algorithm. It selects a specific prediction model based on the historical workload sequence at each prediction point. As presented in Algorithm 1, our model selection method consists of two procedures, one is model selection when the prediction point arrives, and the other is buffer updating when the actual future workload of a historical sequence is available.

The design of our algorithm is guided by two main insights: (1) The best model within a short period tends to stay unchanged (short-term consistency). (2) The sequences that favor the same model have a closer DTW distance. Insight (1) inspires us to leverage recent model selection results for current model selection. Thus, we use a buffer $B$ of length $L$ to store recent and representative model selection results. Considering the limited buffer capacity, we design a score-based buffer management algorithm to update the buffer. We let the buffer store triplets in the form of $\langle sequence, model, score\rangle$. Each triplet records the best prediction $model$ for a specific workload $sequence$, and a $score$. The $score$ measures the universality of such sequence-model matching and its staleness for the current system. For example, $\langle\{1,2,3,4\}, ARIMA, 5\rangle$, indicates the best prediction model for workload sequence $\{1,2,3,4\}$ is ARIMA with a score of 5. The updating process of our buffer is introduced as follows:

*Buffer updating:* As time progresses, the ground truth $e'$ for predicting a historical sequence $e$ will become available. When it's obtained, we run all models in parallel to predict $e$ and select the one with the lowest MSE as $f'$ (line 3). Next, we retrieve the triplet $\tau$ used previously to assist in model selection for $e$ (details will be introduced in the model selection procedure). If the best model $f'$ aligns with the model in $\tau$, it implies that the selection possesses considerable referential significance. Consequently, the $score$ of the $\tau$ will be incremented by $l$ to reflect this heightened relevance. Otherwise, $\tau$ is considered as a misleading term and its $score$ will be reduced by $l$ (lines 4–8). We remove the triplet with the lowest $score$ when the buffer is full (line 9) and add a new triplet into the buffer with $score$ initialized as $L$ (line 10).

Insight (2) further inspires us to leverage DTW distance between current sequence and historical sequences in the buffer for model selection. Combining the triplet buffer, we design the next procedure of our algorithm, namely model selection, and introduce it as follows:

*Model selection:* At each prediction point, we receive current workload sequence $e$ that records the workload changes of the previous time window. To select the prediction model, we calculate the similarity between the current workload sequence and all sequences in the buffer using the DTW distance. We next choose the triplet that contains the sequence with the lowest DTW distance (line 13) to assist our model selection. If the DTW distance between these two sequences is within a threshold $d'$, we select the corresponding model recorded in the triplet to predict the future workload of the current workload sequence. Otherwise, we adopt the same selected model of the closest available prediction point in the past (lines 14–18). At the end, we subtract the score of each triplet by 1 to deduce their staleness. (lines 19–21)

**Algorithm 1:** Model Selection for Workload Prediction.

```
Input: Triplet buffer $B$, model candidates $F$, DTW
  distance threshold $d'$, sequence $e$.
Output: Best prediction model $f$.
 1: Procedure: Buffer updating
 2: Obtain ground truth $e'$ and previous used triplet $\tau$.
 3: Select model $f' = \arg\min_f MSE(f(e), e'), f \in F$.
 4: if $\tau.model = f'$ then
 5:     $\tau.score = \tau.score + l$
 6: else
 7:     $\tau.score = \tau.score - l$
 8: end if
 9: Pop the triplet with the lowest $score$.
10: $B.append((e, f', L))$.
11:
12: Procedure: Model selection
13: Select triplet
      $\tau = \arg\min_\tau DTW(\tau.sequence, e), \tau \in B$
14: if $d < d'$ then
15:     $f = \tau.model$
16: else
17:     $f = f'$
18: end if
19: for all $\tau$ in $B$ do
20:     $\tau.score = \tau.score - 1$
21: end for
```

Our model selection algorithm has two key features. First, low time complexity. For the buffer updating procedure, running inference on all models (line 3) is bottlenecked by the slowest model with max time complexity of $O(|e'|)$ [35]. Our experiment shows that existing typical prediction models take less than 0.1s to infer. Selecting the model with the lowest MSE takes $O(|F|)$, where $|F|$ is size of model set $F$. Popping the triplet with the lowest score takes $O(L)$ (line 9). Thus, this procedure takes at most $O(L + |F| + |e'|)$. For the model selection procedure, it takes $O(L|e|)$ to find the triplet that contains the sequence with lowest DTW distance (line 2), since the FastDTW algorithm [36] only takes $O(|e|)$, where $|e|$ is the length of the sequence $e$. Next, it takes $O(L)$ to update the score of each triplet in the buffer. Thus, this procedure only takes $O(L|e|)$. Second, high generalization ability to models and workload sequences. Our algorithm selects the best model solely based on models' past performance and the workload sequences. It doesn't rely on any model-specific information (e.g., model architecture) or assume specific workload patterns. It can be generalized to any workload prediction model and dynamic workload sequences.

### *C. Hierarchical Reinforcement Learning for Mixed VM Provisioning*

*1) Important Definitions:* Our VM provisioner adopts a mixed VM provisioning strategy and acquires a combination of VMs with different types (spot and on-demand) and configurations (CPU and memory) to serve the later user requests. For serverless providers, the goal is to satisfy the SLOs of serverless workloads while optimizing the provisioning costs. We formally introduce several important definitions for serverless VM provisioning. Assume we have $N$ functions and $M$ different VMs. The first definition is the provisioning costs $c_t$ that the serverless providers need to pay the IaaS providers:

$$c_t = \sum_{m}^{M} x_m^t p_m^t, \tag{3}$$

where $x_m^t$ denotes the number of VM $m$ provisioned at time $t$, $p_m^t$ is the price of $m$ at time $t$.

The second definition is the achieved SLO, which is defined using the following equation:

$$s_t = \frac{1}{N} \sum_{n} \frac{g_n^t}{q_n^t}, \tag{4}$$

where $g_n^t$ denotes the number of invocations for function $n$ served without delay[3] at time $t$, $q_n^t$ denotes the total number of invocations arrives for function $n$ at time $t$. The SLO defined in (4) represents the average percentage of the requests served without delay caused by insufficient resources or VM eviction. This metric also indirectly reflects end-to-end latency, as higher latency typically corresponds to a lower $s_t$. We avoid using end-to-end latency [37] as the SLO due to its impracticality. In large serverless platforms, functions vary in execution times and latency sensitivities, leading to diverse and often unmanageable latency targets [13]. Moreover, neither providers nor users can define reasonable targets for all functions. In contrast, the SLO in (4) enables more effective resource management despite such heterogeneity.

With the above definitions, we formally introduce our VM provisioning problem, which makes online provision decisions to achieve the promised SLO while reducing cost. Our problem is an online decision-making problem involving unknown serverless workload, VM price, and VM eviction rate. What's more, the SLO is affected by not only the provisioning decisions but also the placement of functions. Thus, to analyze its complexity, we make the following two simplifications to our problem: (1) We assume an offline scenario with pre-known workload and the properties of spot VM, including its price and eviction rate. (2) We ignore the potential resource fragmentation and uses the resource-to-demand ratio to approximate the achieved SLO for simplicity. With those, we formulate our VM provisioning problem as follows:

$$\min \quad Cost = \sum_{t}^{T} \sum_{m}^{M} x_m^t p_m^t \tag{5}$$

$$\text{s.t.} \quad \frac{1}{T} \sum_{t} \frac{\sum_m x_m^t r_m^a v_m^t}{w_a^t} \geq S \tag{6}$$

$$\frac{1}{T} \sum_{t} \frac{\sum_m x_m^t r_m^b v_m^t}{w_b^t} \geq S \tag{7}$$

$$x_m^t \in \mathbb{N}, \forall m, t \tag{8}$$

where $r_m^a$ and $r_m^b$ are the CPU and memory capacity of VM $m$ respectively, $v_m^t$ is the eviction rate of VM $m$ at time $t$, $w_a^t$ and $w_b^t$ are the total CPU and memory demand of workloads at time $t$, respectively, and $S$ is the SLO target promised by the serverless provider. Constraints (6) and (7) ensure the SLO is achieved by enforcing a certain resource-to-demand ratio. However, even such a simplified offline version of our provisioning problem is still an NP-hard integer linear programming problem (ILP) [38].

*2) Motivation for a HRL-Based Solution:* The above analysis reveals that even in an idealized offline scenario, our VM provisioning problem remains an NP-hard computational complexity challenge. In real-world production, the challenge intensifies due to dynamic factors such as bursty workloads, fluctuating spot properties, and resource fragmentation from function placement. This complexity renders traditional approaches, like heuristics and rule-based strategies [3], [18], ineffective, as they rely on static assumptions and lack adaptability. They also require extensive expert tuning, limiting scalability and deployment. Classical optimization techniques, such as solvers or convex methods [14], lack the flexibility to respond timely to non-stationary, multi-scale workloads. While DRL algorithms like DDPG [39] and PPO [40] show promise in learning adaptive provisioning policies [26], [41], [42], they struggle in large, high-dimensional action spaces, such as selecting VM types, configurations, and counts [43].

Recall that in Section III-B, we identify that our VM provisioning problem can be solved efficiently by decomposing it into two hierarchical sub-problems, namely VM selection and VM number scaling. Driven by this architecture and deficiency of existing methods, we leverage the current state-of-the-art HRL [44] to solve these two sub-problems efficiently. HRL follows the idea that a task can be more effectively learned and accomplished by breaking it down into a series of subgoals or subtasks. A subtask itself may also be formulated as a Markov Decision Process (MDP) and solved using a standard reinforcement learning algorithm. HRL has proven its superiority in practical problems like goods delivery [45] and task offloading [46]. It offers a promising solution to our mixed VM provisioning problem.

*3) HRL-Based VM Provisioner:* In this part, we propose our hierarchical reinforcement learning (HRL)-based VM provisioner. Following the common practice in RL-based methods to express constraints [15], [47], we define a utility function as the weighted sum of the provisioning costs and the SLO violation, and express it as follows:

$$u_t = c_t + \alpha \min(0, s_t - S), \tag{9}$$

where $\alpha$ is a weight factor adjusting penalty of SLO violations.

Our algorithm employs a higher-level agent to address the VM selection problem over a long-term horizon, and multiple lower-level agents to handle the VM number scaling problem at fine-grained time steps. The design of the higher-level and lower-level agents is described as follows:

*Higher-level Agent Design:* As the price and eviction rate of spot VM change much more slowly than the serverless function, we design a higher-level agent to select VMs with appropriate types and configurations for a relatively long period. We discretize the timeline into multiple fixed periods $p$. The higher-level agent acts at the beginning of each period to solve the VM selection problem. The MDP for the higher-level agent is defined as follows:

- *Episode:* The entire timeline $P$.
- *State $o_p^h$:* Current VM number, information of each VM (eviction rate, CPU and memory configuration, and price), predicted workload, previous utility.
- *Action $a_p^h$:* Select $K$ VMs for provisioning.
- *Reward $r_p^h$:* The sum of the utility in period $p$.

[3]A serverless invocation will be delayed for the following two reasons: (1) No free container is available to serve it. (2) It fails to complete within the grace period after its host VM instance is evicted.

**Algorithm 2:** HRL-Based VM Provisioner Training.

```
Input: Initial higher-level policy, lower-level policy, and
 number of training episodes $E$.
Output: Updated higher-level policy and lower-level
 policies.
 1: Initialize replay buffer $b^h$ for higher-level agent, and
    replay buffers for lower-level agents $[b^l_1, b^l_2, \ldots, b^l_K]$
 2: for episode $e = 1, 2, \ldots, E$ do
 3:     Observe the initial higher-level state $o^h_1$.
 4:     for $p = 1, 2, \ldots, P$ do
 5:         Select higher-level action $a^h_p = \pi(o^h_p)$.
 6:         Observe new higher-level state $o^h_{p+1}$.
 7:         Observe lower-level states $[o^l_{1,1}, .., o^l_{1,K}]$
 8:         for $t = 1, 2, \ldots, T$ do
 9:             Select lower-level action $[a^l_{t,1}, \ldots, a^l_{t,K}]$.
10:             Observe lower-level new state $[o^l_{t,1}, .., o^l_{t,K}]$.
11:             Obtain lower-level reward $[r^l_{t,1}, \ldots, r^l_{t,K}]$.
12:             Store transition $(o^l_{t,k}, a^l_{t,k}, r^l_{t,k}, o^l_{t+1,k})$ in $b^l_k$.
13:             Update each lower-level agent using DQN.
14:         end for
15:         Obtain the higher-level reward $r^h_p$.
16:         Store transition $(o^h_p, a^h_p, r^h_p, o^h_{p+1})$ in $b^h$.
17:         Update higher-level agent using PPO.
18:     end for
19: end for
```

After the higher-level agent selects VMs for the next period, we determine the initial count for each selected VM, as they may differ from the previous period. We first retain the VM counts from the prior selection. For newly selected VMs, we match them to previously selected ones with the closest memory-CPU ratio and calibrate their provisioning to ensure consistent resource allocation.

*Lower-level Agents Design:* We assign each selected VM $k$ a lower-level agent to adjust its quantity. Given the frequent fluctuations in serverless workloads, each period $p$ is further divided into $T$ fine-grained time slots, where lower-level agents operate. These agents dynamically scale the number of selected VMs $K$ as determined by the higher-level agent. The MDP for the $k_{th}$ lower-level agent is defined as follows:

- *Episode:* An entire period $p$ with $T$ time slots.
- *State* $o^l_{t,k}$*:* VM number, information of $k_{th}$ selected VM, previous utility, predicted workload.
- *Action* $a^l_{t,k}$*:* Adjust the VM number by $a\%$, $a \in \{a_{low}, \ldots, 0, \ldots, a_{up}\}$.
- *Reward* $r^l_{t,k}$*:* Utility of current time slot $t$.

The detailed workflow of our HRL-based VM provisioning is illustrated in Fig. 5. At the beginning of each long period, the higher-level agent (red) selects $K$ VMs based on the information of all available types. During this period, each lower-level agent (blue) adjusts the number of its assigned VM at each fine-grained time slot. The two levels cooperate hierarchically to meet the SLO while minimizing costs. Although theoretically our problem can be addressed with classical RL algorithms (e.g., PPO [40]) by exploring both VM selection and VM autoscaling simultaneously, these approaches often fail in a large action space. Empirically, we find that PPO without decomposition

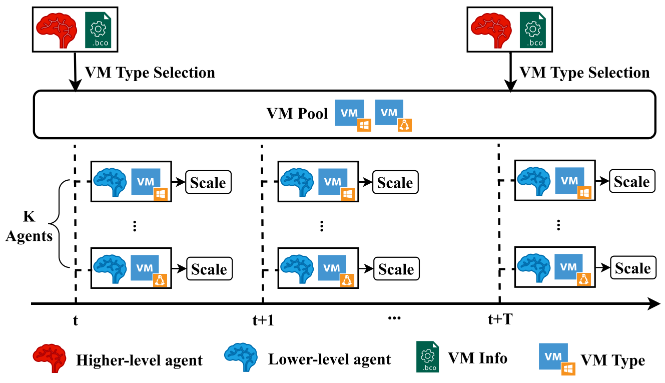


Fig. 5. HRL-based mixed VM provisioning.

incurs 2.5 times of provisioning cost and 4.4 times of SLO violation rate compared to HRL due to the massive action space.

In real-world scenarios, both the VM selection and VM number scaling can introduce latency, which mainly arises from the time required for decision-making and the operational overhead involved. In terms of decision-making latency, we lately will show that our method incurs negligible millisecond-level delays. For operation overhead, this form of latency can be effectively mitigated by letting the HRL agents make the predictive decision beforehand for pre-allocation [15]. The extent of this preliminary allocation can be dynamically adjusted to suit providers' computing clusters.

The training procedure of our HRL-based provisioner is presented in Algorithm 2. It consists of three nested loops: a main loop for episode initialization (lines 2–19), an outer loop for training the higher-level agent (lines 4–18), and an inner loop for training the lower-level agents (lines 8–14). We begin by initializing replay buffers for both agent levels (line 1). The main loop initializes each episode. In the outer loop, the higher-level agent interacts with the environment and selects VMs. Then, in the inner loop, each selected VM is assigned a lower-level agent to scale its count. Transitions are stored in each lower-level agent's buffer, and training is performed using Deep DQN [48] with a shared reward. DQN is chosen for its simplicity and proven effectiveness in multi-agent scenarios [47]. After the inner loop, the transition is stored in the higher-level agent's buffer, and its policy is updated using PPO [40]. PPO is selected for its sample efficiency, as the higher-level agent operates at a coarser time scale and receives fewer samples. Training continues until all episodes are completed.

## D. Function Collocation Placement With Mixed VMs

After determining the provisioning plan, the next challenge is how to efficiently place the function containers onto the provisioned instances. A poor placement strategy may lead to severe resource fragmentation and decreased resource utilization, even in simpler environments with homogeneous VMs [13]. In contrast, an efficient placement strategy can not only improve resource utilization but also provide more space for our HRL-based provisioner to explore cost-reducing provisioning decisions.

In this section, we present an effective function collocation placement strategy to improve resource utilization. Our key idea is to collocate resource-complementary functions so that the resource ratio of the collocation closely matches the resource ratio of the VM, thereby minimizing resource fragmentation within each VM. OWL [13] adopts a similar idea, but it mainly focuses on mitigating resource contention and only applies to homogeneous VMs with the same configuration. Our method jointly considers the VM capacity and function resource requirement and is applicable for function placement on heterogeneous VM instances.

The details of our function placement algorithm is shown in Algorithm 3. We start by considering a simple offline function placement scenario, where we are given $M$ selected VMs and $N$ functions. Each VM has multiple empty instances and each function have multiple containers waiting for placement. The online version involves slight modifications and will be detailed shortly. To identify resource-complementary functions, we first define the resource ratio of a function as the ratio of its CPU requirement to memory requirement. The resource ratio $r_{i,j}$ of a collocated function pair with functions $i$ and $j$ can be calculated similarly using the sum of the CPU and memory of two functions. Next, we maintain a two-dimensional collocation matrix $C$ (line 1), with each entry in $C$ defined as follows:

$$C_{ij} = [a_{ij}, b_{ij}, h_{ij}], \tag{10}$$

where $a_{ij}$ and $b_{ij}$ represent the CPU and memory usage of the function pair ($i$, $j$), respectively, and $h_{ij}$ is the minimum number of containers between $i$ and $j$, representing the maximum number of container pair that can be used for collocation. After that, we define the dominant collocation resource ratio $r_d$ for unplaced functions as the weighted sum of all entries in $C$ (line 2):

$$r_d = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} h_{ij} \frac{a_{ij}}{b_{ij}}, \tag{11}$$

which represents the collocation resource ratio of the majority of the unplaced containers. We then select the VM that has the closest resource ratio $r_m$ to $r_d$ (line 4), so that we can place as many functions as possible while ensuring high resource utilization. We sort the entries in $C$ according to the distance of resource ratio between the function pair and selected VM, and keep placing them into the instances in order (lines 5–8). The above procedure is repeated for $K$ steps (lines 3–10). When the collocation process is done, we place the remaining functions using the First-Fit algorithm (line 11). In addition, we reduce the impact of eviction by collocating long-running functions at on-demand VMs in prior.

Algorithm 3 assumes an offline scenario where the number of function containers and VM instances is fixed [13]. In contrast, in online settings, these numbers fluctuate with dynamic serverless workloads. Replacing containers due to such changes may introduce significant overhead and degrade SLOs. To address this, we propose three techniques to reduce placement overhead: (1) Provisioning-aware replacement: Scaling VM counts has less impact on placement than changing VM types, as configurations remain unchanged. Thus, we re-run the algorithm only when the higher-level agent changes the VM selection strategy, typically once per hour. (2) Tag-assisted placement: Since selected VMs remain constant during scaling, previous collocation results can be reused. We attach a tag containing the function ID and instance ID of a removed container to guide placement of the new one onto the same or a similar instance (based on resource ratio).

**Algorithm 3:** Function Collocation Placement.

```
Input: $M$ VMs and $N$ functions.
Output: Function container placement plan.
 1: Initialize a collocation matrix $C = \mathbb{R}^{N\times N}$.
 2: Obtain dominant collocation resource ratio $r_d$ by (11).
 3: for m = 1,2,...,M do
 4:     Select VM $m = \arg\min_m |r_m - r_d|$.
 5:     while $hasIdleVM(m)$ & $hasUnplacedFunc()$ do
 6:         Select function pair
            $(i, j) = \arg\min_{i,j} |r_{i,j} - r_m|$
 7:         Place $(i, j)$ into $m$'s instances exhaustively.
 8:     end while
 9:     Update $C$ and $r_d$.
10: end for
11: Place the remaining functions using First-Fit.
```

If no tag is available, First-Fit is used. (3) Selective collocation: Collocation complexity mainly arises from the collocation matrix. By targeting the top 10% of functions that account for 99% of the workload [4], we reduce matrix size. First-Fit is applied to the remaining functions.

### E. Discussion and Future Work

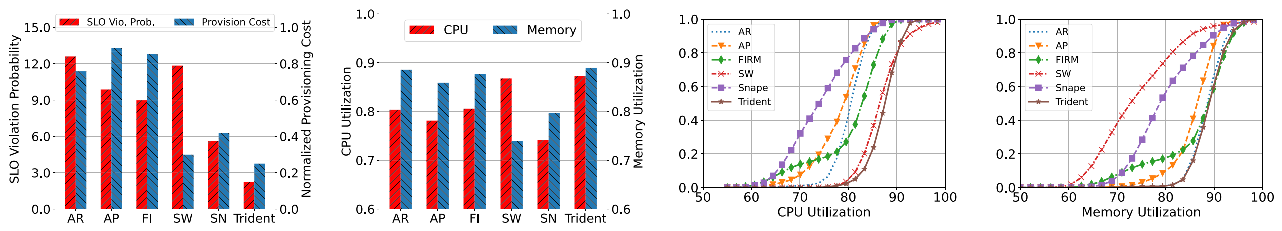
While Trident demonstrates significant improvements in workload prediction, VM provisioning, and function placement, there are still several limitations to address. First, Trident does not explicitly tackle the cold-start latency issue, which is a critical challenge in serverless platforms, especially under bursty or unpredictable workloads. Second, the HRL-based provisioner may still face scalability challenges in extremely large-scale or highly dynamic environments. In our future work, we plan to explore cold-start-aware provisioning and pre-warming strategies, improve the scalability of Trident with distributed training, and adapt the framework to work alongside Kubernetes-native autoscalers for broader deployment compatibility and operational transparency.

## V. Experimental Evaluation

### A. Experiment Setup

*Dataset:* We use Azure Function 2019 [10] to evaluate our approach. It records the function invocation patterns within 14 days, together with the distribution of execution times per function and the distribution of memory usage per application. We sample 1 K functions from the dataset. We allocate the application-level memory usage to each function proportional to their average invocation times and obtain the function-level memory usage. The CPU requirement of each function is generated by multiplying the memory requirement with the feasible memory-CPU ratio reported in [49]. We simulate the dynamics of spot VMs using the SpotWeb dataset [14]. We select 20 unique VMs from the dataset for experiments and half of the VMs are spot VMs. The configurations of these VMs range from 4-48 in terms of CPU cores, and 15-192 GBs in terms of memory capacity.

*Experiment environment and parameter settings:* Our experiment environment is built upon a Ubuntu 20.04 machine with an Intel I7-12700 CPU and an RTX 2070 SUPER GPU. The hyper-parameters of the model selection module are obtained



(a) SLO violations probability and normalized provisioning costs.

(b) Average CPU and memory utilization.

(c) Over time CPU utilization CDF.

(d) Over time memory utilization CDF.

Fig. 6. End-to-end performance comparison.

offline using grid search. The interval for workload prediction is three minutes and we use the workload of the previous 30 minutes to predict the next 15 minutes. We implement the HRL-based VM provisioning module using PyTorch with Python 3.7. For the higher-level agent, the decision step is 1 h, the network architecture is composed of three fully connected layers with 400 hidden units and ReLU as the activation function. The number of lower-level agents is set to 6. For lower-level agents, the decision step is 3 minutes, the network structure is composed of three fully connected layers with 128 hidden units and ReLU as the activation function. We optimize both the higher-level agent and lower-level agents using Adam with a learning rate of 0.0001 and a batch size of 32.

*Evaluation metric:* We use MSE to evaluate our model selection method. For end-to-end evaluations, we use provisioning costs and SLO violations probability. The latter one is calculated by subtracting the achieved SLO defined in (4) from the SLO target. Trident can be flexibly adapted to other SLOs [37] by training on other utility functions.

*Baselines:* For end-to-end comparisons, we evaluate Trident with the following 5 baselines:

- *AWS Reactive Scaling (AR) [17]:* AR is AWS's default scaling strategy. It scales the resource based on pre-defined thresholds and real-time metrics monitoring.
- *AWS Predictive Scaling (AP) [16]:* AP is AWS's advanced strategy uses machine learning models (i.e., DeepAR) to predict workloads for resource scaling.
- *SpotWeb (SW) [14]:* SW is a mixed VM purchasing method based on multi-period portfolio optimization. SpotWeb considers heterogeneous VMs.
- *FIRM (FI) [26]:* FI uses DDPG to adjust the amount of various kinds of resources for microservices.
- *Snape (SN) [12]:* SN is a constrained RL-based VM provisioning approach that dynamically mixes on-demand and spot VMs with the same configuration.

For workload prediction, we compare our dynamic model selection algorithm with the following six widely-used prediction models and a model ensemble approach:

- Two statistical models, ARIMA (AR) and TBATS (TB) [25]. They are used by Alibaba [24] for flow prediction.
- Two machine learning-based models, Support Vector Regression (SVR) [27] and XGBoost (XG) [28]. SVR is a kernel-based algorithm for regression. XG is a decision tree-based method.
- Two deep learning-based models, DeepAR (DAR) [23] and Transformer (TF) [50]. DAR is an LSTM-based model adopted by Alipay [15]. Transformer applies attention mechanism and is used by Azure [12].
- Averaging Ensemble (EN): an ensemble method that averages the prediction of different models.

## B. End-to-End Performance

Fig. 6(a) presents the end-to-end performance of Trident and five baselines. Trident achieves the lowest SLO violation probability and provisioning cost simultaneously. Its SLO violation rate is only 2.3%, compared to 12.6%, 9.9%, 9.0%, 11.8%, and 5.6% for AR, AP, FI, SW, and SN, respectively. AR performs the worst, as its reactive strategy struggles with fluctuating serverless workloads. AP improves SLO satisfaction by about 3% through workload prediction. SW also uses prediction, but its portfolio-based heuristics result in risky provisioning and higher SLO violations. FI and SN apply reinforcement learning and perform relatively better, yet both ignore VM dynamics and combinations, limiting their effectiveness. In contrast, Trident adapts well to workload fluctuations and delivers reliable SLO guarantees.

Next, we examine the provisioning costs of each method. Compared to the baselines, Trident reduces costs by 67.01%, 71.80%, 70.64%, 16.47%, and 40.12%, respectively. While SW achieves a comparable cost, it suffers from a high SLO violation rate (nearly 12%), suggesting under-provisioning or overly aggressive use of spot VMs. Although SN supports mixed provisioning of on-demand and spot VMs, it consistently uses spot VMs of the same type, which is neither cost-effective nor resource-efficient due to dynamic spot prices and heterogeneous function requirements. In contrast, Trident dynamically identifies reliable, low-cost VMs and adjusts provisioning strategies based on spot price and eviction rate variations, resulting in the lowest overall provisioning cost.

Fig. 6(b) shows the CPU and memory utilization of each method. Trident achieves the highest CPU and memory utilization at 87.27% and 88.96%, respectively. For the five baselines, CPU utilization is 80.36%, 78.12%, 80.57%, 86.76%, and 74.15%, while memory utilization is 88.56%, 85.89%, 87.63%, 73.02%, and 79.69%. Thanks to its mixed provisioning strategy and effective function collocation, Trident maintains balanced resource usage, with less than a 3% gap between CPU and memory utilization. As shown in Fig. 6(c) and (d), Trident sustains a higher and more concentrated region of resource usage, indicating its ability to respond swiftly to workload fluctuations and produce efficient provisioning plans.

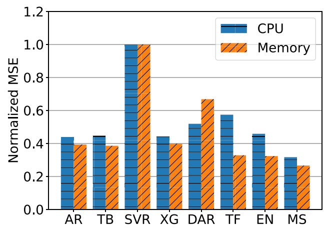


Fig. 7. Prediction performance comparison.

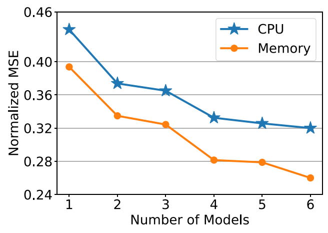


Fig. 8. Prediction with the different number of models.

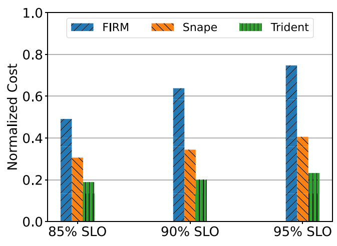


Fig. 9. Provisioning costs under different SLO targets.

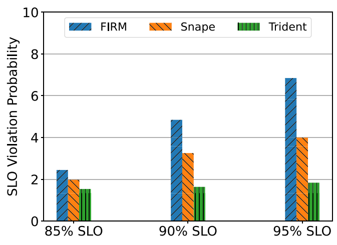


Fig. 10. SLO violations probability under different SLOs.

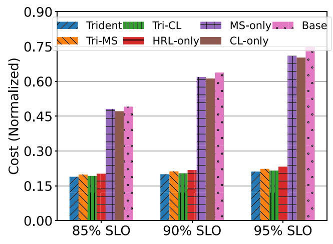


Fig. 11. Normalized provisioning costs for ablation study.

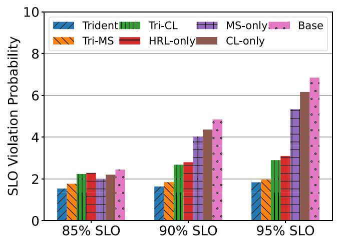


Fig. 12. SLO violations probability for ablation study.

## *C. Workload Prediction Performance*

Fig. 7 shows the performance of our adaptive model selection algorithm. The normalized MSE for memory prediction is 0.44, 0.44, 1.00, 0.44, 0.52, 0.58, 0.46, and 0.32, and for CPU prediction is 0.40, 0.38, 1.00, 0.40, 0.66, 0.32, 0.32, and 0.26, for AR, TB, SVR, XG, DAR, TF, EN, and MS respectively. Our method MS achieves the lowest MSE on both tasks. Among individual models, TF performs best on memory prediction, and AR on CPU prediction. SVR performs the worst on both. Compared to the worst single model, our method reduces MSE by up to 68% (CPU) and 74% (memory); compared to the best single model, it achieves reductions of 19% and 28%, respectively.

We next examine how the number of prediction models affects the performance of our model selection algorithm. We add the models incrementally in the order of AR, TB, SVR, XG, DAR, and TF. As shown in Fig. 8, the prediction performance improves monotonically for both CPU and memory tasks as more models are included. This demonstrates the scalability of our method, which continues to benefit from additional models. Such scalability gives the algorithm long-term potential to evolve by incorporating more advanced predictors over time.

## *D. Sensitivity Analysis on Different SLO Targets*

Given the heterogeneity of serverless functions and cost considerations, providers may wish to adjust SLO targets for different workloads. To evaluate Trident under varying SLO requirements, we test it with three levels: 85%, 90%, and 95%, where the percentage indicates the proportion of requests served without delay. These targets are designed based on the 95% latency threshold commonly used in prior work [51], with 5% decrements to create lower levels. While other SLO targets can also be supported, we focus on these three as representative cases. Since the three heuristic baselines do not explicitly optimize for SLO targets, we compare only the RL-based methods. As shown in Figs. 9 and 10, Trident consistently achieves the lowest SLO violation rates, 1.53%, 1.64%, and 1.84%, and the lowest normalized provisioning costs, 0.18, 0.23, and 0.23 across all SLO levels. These results demonstrate Trident's adaptability and its effectiveness in balancing SLO guarantees with cost efficiency.

## *E. Ablation Study*

In Figs. 11 and 12, we analyze the effectiveness of three modules in Trident, namely the dynamic model selection module, the HRL-based VM provisioning module, and the collocation function placement module. We choose ARIMA, FIRM, and First-Fit as our baselines, corresponding to three modules in Trident. In addition to Trident, which includes all three modules, we create five other benchmarks, Tri-MS (replace function placement module of Tri-Full with First-Fit), Tri-CL (replace model selection module of Tri-Full with ARIMA), HRL-only

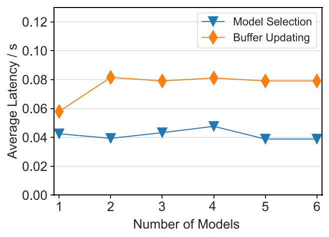


Fig. 13. Overhead for model selection module.

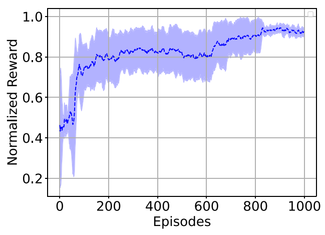


Fig. 14. Learning progress of VM provisioning module.

(HRL-based provisioner with ARIMA and First-Fit), MS-only (integrate model selection module with FIRM), CL-only (integrate function placement module with FIRM). Each benchmark is evaluated in an end-to-end manner under different SLO targets.

Generally speaking, Tri-Full performs better in both SLO satisfaction and provisioning costs savings compared to the other three benchmarks. In contrast, HRL-Only has the highest SLO violations probability and the highest provisioning costs. Comparing HRL-Only and Tri-MS, we can see that adaptive model selection reduces the SLO violations probability by 29.01%, 34.01%, and 36.52% at 85%, 90%, 95% SLO targets, respectively. Comparing HRL-Only and Tri-CL, we can see that it reduces the provisioning costs by 4.67%, 6.42%, and 7.27% under 85%, 90%, and 95% SLO, respectively. This indicates that the collocation placement module contributes more to cost savings.

We next evaluate the effectiveness of each module when integrated with the baseline. Comparing HRL-only and Base, our HRL-based provisioner outperforms FIRM's native DDPG in both SLO attainment and cost savings, especially under strict SLO targets. Comparing MS-only and Base, the model selection module reduces SLO violations by 19.22%, 17.41%, and 22.36% under three SLO targets. This shows that our model selection module can effectively assist other provisioning strategies. When integrated with FIRM, the collocation placement module reduces provisioning costs by 4.11%, 4.98%, and 6.38% under three SLO targets, demonstrating its adaptability to existing baselines. However, we observe that the improvements from these two modules are more limited under FIRM than under our HRL-based provisioner, indicating that HRL can better exploit accurate workload prediction and improved resource utilization to optimize system performance.

### *F. Overhead Analysis*

This section analyzes the runtime overhead of Trident's three modules. For the model selection module, we examine the relationship between runtime latency and the number of prediction models. As shown in Fig. 13, the latency remains stable under 60 milliseconds, regardless of model count. This stability is due to the fact that the main latency source is the DTW distance computation between the current workload and sequences stored in a fixed-length buffer, making latency independent of model quantity. Regarding the buffer update procedure, latency increases slightly from one to two models but remains stable thereafter. Since all model predictions are executed in parallel, the bottleneck is determined by the model with the highest inference latency. These results demonstrate the scalability of our dynamic model selection module.

Next, we examine the overhead of our HRL-based VM provisioning module. The training process is visualized in Fig. 14. It takes less than 1000 episodes to converge and about 2 hours to finish on our machine. We observe no instability issues during the training. For inference overhead, it consists of two parts: inference latency of the higher-level agent (about 50 ms) and that of the lower-level agents (about 15 ms). Both are negligible in the context of online provisioning. The higher-level agent incurs slightly higher latency due to its larger neural network, which processes information from all VMs. These results indicate that our provisioning module can respond promptly to workload changes and operate effectively in dynamic serverless environments.

For the function placement module, the process of running the whole function collocation placement in Algorithm 3 takes about 1 seconds to accomplish in our experiment setting. Though the overhead is relatively larger compared to the model selection module and VM provisioning module, this process only takes place when the upper-level agent changes the VM selection strategy, whose frequency is more than an hour as we've mentioned in Section IV-D. In addition, we simply use the recorded tags or First-Fit in most cases, which only took negligible milliseconds to accomplish. In conclusion, our function placement module can efficiently perform real-time function placement. Additional generalization results are provided in our supplementary material.

## VI. Related Work

### *A. Cloud Workload Prediction*

Workload prediction plays a guiding role in cloud resource management. Different models have been proposed to predict the future workload of either an individual application or a large cluster. For example, ARIMA and cubic spine regression [25] are employed in [14] for spot VM purchasing. Amazon also deploys well-trained machine learning models for scaling EC2 instances [16]. In [29], the random forest algorithm with Bayesian optimization is employed to predict the workload of serverless data analysis jobs. Nonetheless, these works typically employ a single off-the-shelf model for workload prediction, which can result in significant prediction errors. To address this problem, we propose a novel fine-grained model selection scheme that leverage the power of multiple models. It can offer a more accurate prediction with negligible overhead.

### *B. VM Provisioning and Auto-Scaling*

To serve continuous requests or latency-sensitive applications, cloud providers are required to provision a set of VMs beforehand. Due to the reliability of on-demand VMs, efficient on-demand VM provisioning has been widely studied [18], [52], [53]. Researchers from Azure [18] present a heuristic search-based method for predictive VM provision. In [52], two solutions based on control theory and queuing theory are proposed to make autoscaling decisions for cloud applications. COUNSEL [53] applies DRL to handle the dynamic workloads and efficiently manages the configurations of an arbitrary multi-component service. To further reduce the VM provisioning costs, a fair amount of work also explores the potential of incorporating spot VM (or transient VM) into VM provision [12], [14], [54]. For example, Tributary [54] builds models of preemption likelihood and selects collections of resource allocations that satisfy SLO requirements. However, the heuristics-based methods [18], [54] fail to handle the dynamics and complexity of mixed VM provisioning. DRL-based methods like COUNSEL [53] focus on small-scale scenarios involving single-application VM selection, without determining the actual number of VM instances, which is critical in provider-oriented serverless computing. By comparison, Trident incorporates a tailored HRL-based approach that intelligently selects cost-effective and reliable VMs and makes auto-scaling decisions.

## VII. Conclusion

In this paper, we present Trident, a provider-oriented resource management framework for serverless computing. Trident tackles three critical problems for serverless resource management, namely workload prediction, VM provisioning, and function placement. We first propose a dynamic model selection scheme to improve the accuracy of serverless workload prediction. With the accurately predicted workload, we then design an HRL-based VM provisioner to perform cost-effective and SLO-guaranteed VM provisioning. To increase resource utilization and further reduce the provisioning costs, we design a function collocation placement algorithm for heterogeneous VMs that collocates resource-complementary function pairs into VM instances. Experiments on real-world serverless workloads demonstrate the superiority of our system compared with other industrial and academic solutions. We have provided public access to our core code at https://github.com/oximi123/Trident#.

## References

[1] E. Jonas et al., "Cloud programming simplified: A berkeley view on serverless computing," 2019, *arXiv: 1902.03383*.
[2] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," *IEEE Trans. Serv. Comput.*, vol. 16, no. 2, pp. 1522–1539, Mar./Apr. 2023.
[3] AWS, "AWS lambda," 2023. [Online]. Available: https://aws.amazon.com/lambda/
[4] M. Azure, "Azure functions," 2023. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/
[5] OpenWisk, "Openwisk," 2023. [Online]. Available: https://openwhisk.apache.org/
[6] Fission, 2023. [Online]. Available: https://fission.io/
[7] S. Bhardwaj, L. Jain, and S. Jain, "Cloud computing: A study of infrastructure as a service (IaaS)," *Int. J. Eng. Inf. Technol.*, vol. 2, no. 1, pp. 60–63, Jun. 2010.
[8] "Netlify: Connect everything. Build anything," 2025. [Online]. Available: https://www.netlify.com/
[9] Y. Zhang et al., "Faster and cheaper serverless computing on harvested resources," in *Proc. ACM SIGOPS Symp. Operating Syst. Princ.*, 2021, pp. 724–739.
[10] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 205–218.
[11] J. Gao, H. Wang, and H. Shen, "Machine learning based workload prediction in cloud computing," in *Proc. Int. Conf. Comput. Commun. Netw.*, 2020, pp. 1–9.
[12] F. Yang et al., "Snape: Reliable and low-cost computing with mixture of spot and on-demand VMs," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 631–643.
[13] H. Tian et al., "OWL: Performance-aware scheduling for resource-efficient function-as-a-service cloud," in *Proc. Symp. Cloud Comput.*, 2022, pp. 78–93.
[14] A. Ali-Eldin et al., "Spotweb: Running latency-sensitive distributed web services on transient cloud servers," in *Proc. Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2019, pp. 1–12.
[15] S. Xue et al., "A meta reinforcement learning approach for predictive autoscaling in the cloud," in *Proc. ACM SIGKDD Conf. Knowl. Discov. Data Mining*, 2022, pp. 4290–4299.
[16] AWS, "AWS predictive scaling," 2023. [Online]. Available: https://aws.amazon.com/cn/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/
[17] AWS, "AWS auto scaling," 2023. [Online]. Available: https://aws.amazon.com/cn/ec2/autoscaling/
[18] C. Luo et al., "Intelligent virtual machine provisioning in cloud computing," in *Proc. Int. Joint Conf. Artif. Intell.*, 2021, pp. 1495–1502.
[19] Google, "Google cloud functions," 2023. [Online]. Available: https://cloud.google.com/functions/
[20] M. Zhang, Y. Zhu, C. Zhang, and J. Liu, "Video processing with serverless computing: A measurement study," in *Proc. ACM Workshop Netw. Operating Syst. Support Digit. Audio Video*, 2019, pp. 61–66.
[21] J. Jiang et al., "Towards demystifying serverless machine learning training," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 857–871.
[22] V. Shankar et al., "Serverless linear algebra," in *Proc. ACM Symp. Cloud Comput.*, 2020, pp. 281–295.
[23] D. Salinas, V. Flunkert, J. Gasthaus, and T. Januschowski, "DeepAR: Probabilistic forecasting with autoregressive recurrent networks," *Int. J. Forecasting*, vol. 36, no. 3, pp. 1181–1191, Jul. 2020.
[24] Q. Wen et al., "RobustPeriod: Robust time-frequency mining for multiple periodicity detection," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 2328–2337.
[25] R. Hyndman, A. B. Koehler, J. K. Ord, and R. D. Snyder, *Forecasting With Exponential Smoothing: The State Space Approach*. Berlin, Germany: Springer, 2008.
[26] H. Qiu et al., "FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 805–825.
[27] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statist. Comput.*, vol. 14, pp. 199–222, Aug. 2004.
[28] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, Aug. 2016, pp. 785–794.
[29] A. D. Mohapatra and K. Oh, "Smartpick: Workload prediction for serverless-enabled scalable data analytics systems," in *Proc. Int. Middleware Conf.*, 2023, pp. 29–42.
[30] L. Wang et al., "Peeking behind the curtains of serverless platforms," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 133–146.
[31] H. Zhang, Y. Tang, A. Khandelwal, and I. Stoica, "SHEPHERD: Serving DNNs in the wild," in *Proc. USENIX Symp. Networked Syst. Des. Implementation*, 2023, pp. 787–808.
[32] AWS, Tutorial: Get started with amazon EC2 windows instances. 2023. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/EC2_GetStarted.html
[33] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *Proc. 3rd Int. Conf. Knowl. Discov. Data Mining*, 1994, pp. 359–370.
[34] AWS, "AWS spot instance," 2023. [Online]. Available: https://docs.aws.amazon.com/zh_cn/AWSEC2/latest/WindowsGuide/using-spot-instances.html/
[35] H. Zhou et al., "Informer: Beyond efficient transformer for long sequence time-series forecasting," in *Proc. AAAI Conf. Artif. Intell.*, 2021, pp. 11106–11115.
[36] S. Salvador and P. Chan, "Toward accurate dynamic time warping in linear time and space," *Intell. Data Anal.*, vol. 11, no. 5, pp. 561–580, Jun. 2007.
[37] Z. Wen, Y. Wang, and F. Liu, "StepConf: SLO-aware dynamic resource configuration for serverless function workflows," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1868–1877.

[38] L. A. Wolsey and G. L. Nemhauser, *Integer and Combinatorial Optimization*. Hoboken, NJ, USA: Wiley, 1999.
[39] T. P. Lillicrap et al., “Continuous control with deep reinforcement learning,” 2015, *arXiv:1509.02971*.
[40] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017, *arXiv: 1707.06347*.
[41] C. Ling, K. Peng, S. Wang, X. Xu, and V. C. M. Leung, “A multi-agent DRL-based computation offloading and resource allocation method with attention mechanism in MEC-enabled IIoT,” *IEEE Trans. Serv. Comput.*, vol. 17, no. 6, pp. 3037–3051, Sep. 2024.
[42] H. Bai et al., “DRPC: Distributed reinforcement learning approach for scalable resource provisioning in container-based clusters,” *IEEE Trans. Serv. Comput.*, vol. 17, no. 6, pp. 3473–3484, Nov./Dec. 2024.
[43] G. Dulac-Arnold et al., “Deep reinforcement learning in large discrete action spaces,” 2015, *arXiv:1512.07679*.
[44] S. Pateria, B. Subagdja, A.-H. Tan, and C. Quek, “Hierarchical reinforcement learning: A comprehensive survey,” *ACM Comput. Surv.*, vol. 54, no. 5, pp. 1–35, Jun. 2021.
[45] Y. Ma et al., “A hierarchical reinforcement learning based optimization framework for large-scale dynamic pickup and delivery problems,” in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2021, pp. 23609–23620.
[46] C. Sun, X. Li, C. Wang, Q. He, X. Wang, and V. C. M. Leung, “Hierarchical deep reinforcement learning for joint service caching and computation offloading in mobile edge-cloud computing,” *IEEE Trans. Serv. Comput.*, vol. 17, no. 4, pp. 1548–1564, Jul./Aug. 2024.
[47] B. Zhu, S. Lin, Y. Zhu, and X. Wang, “Collaborative hyperspectral image processing using satellite edge computing,” *IEEE Trans. Mobile Comput.*, vol. 23, no. 3, pp. 2241–2253, Mar. 2024.
[48] V. Mnih et al., “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, Jun. 2015, Art. no. 529.
[49] Alibaba, “Alibaba pay-as-you-go billing method,” 2023. [Online]. Available: https://www.alibabacloud.com/help/en/function-compute/latest/pay-as-you-go#concept-2557381
[50] A. Vaswani et al., “Attention is all you need,” in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
[51] A. Mahgoub et al., “ORION and the three rights: Sizing, bundling, and prewarming for serverless dags,” in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 303–320.
[52] G. Quattrocchi, E. Incerto, R. Pinciroli, C. Trubiani, and L. Baresi, “Autoscaling solutions for cloud applications under dynamic workloads,” *IEEE Trans. Serv. Comput.*, vol. 17, no. 3, pp. 804–820, May/Jun. 2024.
[53] A. Hegde, S. G. Kulkarni, and A. S. Prasad, “Counsel: Cloud resource configuration management using deep reinforcement learning,” in *Proc. Int. Symp. Cluster Cloud Internet Comput.*, 2023, pp. 286–298.
[54] A. Harlap et al., “Tributary: Spot-dancing for elastic services with latency SLOs,” in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 1–14.

**Botao Zhu** received the BEng degree from Software College, Northeastern University, China, in 2022. He is currently working toward the PhD degree with Global College, Shanghai Jiao Tong University. His research interests include machine learning systems, cloud, and edge computing.

**Yifei Zhu** (Member, IEEE) received the BE degree from Xi’an Jiaotong University, China, in 2012, the MPhil degree from the Hong Kong University of Science and Technology, China, in 2015, and the PhD degree in computer science from Simon Fraser University, Canada, in 2020. He is currently an associate professor with Global College, Shanghai Jiao Tong University, China. His research interests include edge computing, multimedia networking, and distributed machine learning systems. He is an associate editor of *IEEE Internet of Things Journal*, and has served as the area chair, and organization chair for several IEEE/ACM conferences.

**Chen Chen** (Member, IEEE) received the BEng degree from Tsinghua University, in 2014, and the PhD degree from the Hong Kong University of Science and Technology, in 2018. He is an associate professor with John Hopcroft Center for Computer Science, Shanghai Jiao Tong University. His recent research interests include distributed deep learning, large language model systems, and networking. He previously worked as a researcher with the Theory Lab, Huawei Hong Kong Research Center.

**Linghe Kong** (Senior Member, IEEE) received the PhD degree in computer science from Shanghai Jiao Tong University, in 2013. He is a professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. Before that, he was a postdoctoral researcher with Columbia University, McGill University, and Singapore University of Technology and Design. His research interests include wireless networks, Big Data, mobile computing, and Internet of things.