Incendio: Priority-Based Scheduling for Alleviating Cold Start in Serverless Computing

Xinquan Cai ⓘ, Qianlong Sang ⓘ, Chuang Hu ⓘ, *Member, IEEE*, Yili Gong ⓘ, Kun Suo ⓘ,
Xiaobo Zhou ⓘ, *Senior Member, IEEE*, and Dazhao Cheng ⓘ, *Senior Member, IEEE*

Abstract—In serverless computing, cold start results in long response latency. Existing approaches strive to alleviate the issue by reducing the number of cold starts. However, our measurement based on real-world production traces shows that the minimum number of cold starts does not equate to the minimum response latency, and solely focusing on optimizing the number of cold starts will lead to sub-optimal performance. The root cause is that functions have different priorities in terms of latency benefits by transferring a cold start to a warm start. In this paper, we propose *Incendio*, a serverless computing framework exploiting priority-based scheduling to minimize the overall response latency from the perspective of cloud providers. We reveal the priority of a function is correlated to multiple factors and design a priority model based on Spearman's rank correlation coefficient. We integrate a hybrid Prophet-LightGBM prediction model to dynamically manage runtime pools, which enables the system to prewarm containers in advance and terminate containers at the appropriate time. Furthermore, to satisfy the low-cost and high-accuracy requirements in serverless computing, we propose a Clustered Reinforcement Learning-based function scheduling strategy. The evaluations show that Incendio speeds up the native system by 1.4×, and achieves 23% and 14.8% latency reductions compared to two state-of-the-art approaches.

Index Terms—Serverless computing, cold start, priority, prediction, scheduling, in-memory computing, distributed systems.

I. INTRODUCTION

SERVERLESS computing expands on state-of-the-art cloud computing by releasing developers of serverless applications from capacity configuration, management, maintenance, fault tolerance, and scaling of containers, VMs, or physical servers. Owing to the advantages of high maintainability and flexibility, commercial giants, like AWS Lambda [1], Google Cloud Functions [2], and Microsoft Azure Functions [3], have all deployed serverless computing services on their platforms. Open-source serverless computing platforms, e.g., Apache OpenWhisk [4] and OpenFaaS [5], are also springing up in the community. In addition, serverless computing promotes Functions as a Service (FaaS), a cloud service model that is typically associated with building microservices applications and allows users to execute code in response to events without the complex infrastructure, to become increasingly popular among developers and users.

In FaaS, a function is served by one or multiple containers. When a function request comes in, the FaaS system will check whether there is a container ready to serve the invocation. When an idle container is already available, we call it a warm container (*warm start*). If there is no container ready, the function will spin up a new one (*cold start*). Since the cold start entails creating a container, fetching and installing necessary libraries and dependencies before the function itself can be executed, cold start latency is often several or even tens of times higher than a warm start [6], [7]. For instance, as shown in Fig. 1, the cold start latency of `HelloWorld` function implemented by different programming languages can reach up to 80× longer than warm start latency, which will result in extremely slow applications and services response. Due to the unacceptable delays for real-time online services, users prefer providers who can guarantee response time in tens of milliseconds [7], [8].

In recent years, researchers have put a great deal of effort into addressing the cold start problem in FaaS. Techniques have been developed to keep the container live in the memory for future invocations to avoid unnecessary initialization, which is known as keep-alive policy [9], [10], [11]. However, these techniques often incur an unacceptable cost of keeping the runtime alive [12], [13], [14]. In comparison, others have proposed to prepare the live runtime environment in advance to eliminate the occurrence of cold start, known as prewarm policy [10], [15], [16]. Nevertheless, it is difficult to determine when and what to prewarm in production environments. Many efforts have
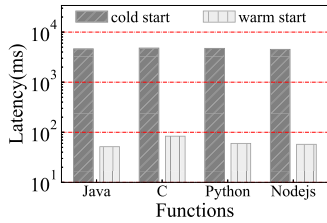
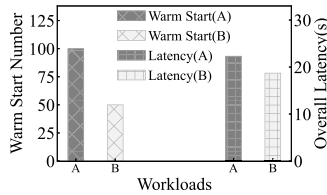Fig. 1.   Latency: cold start *vs.* warm start.

Fig. 2.   The number of warm starts and overall latency.

introduced certain randomness in predicting prewarm functions. Recently, methods such as CheckPoint/Recovery acceleration [17], [18], [19], are proposed to accelerate runtime startup or state recovery.

However, all the innovations in the literature focused on reducing the total number of cold starts. Instead, the occurrence of a cold start can lead to a higher latency reduction benefit in some cases. We measure the latency benefit by adding the latency difference between the cold start and the warm start for all requests hitting the warm start. In addition, although the number of cold starts positively correlates with the overall latency in most cases, we find that the differences and priorities of the services cannot be ignored. Specifically, we measure and compare the latency difference between the cold start and the warm start of various requests. Fig. 2 shows the number of cold starts and the overall latency of the measurement on two different scheduling strategies over Azure public dataset [10], [11], where strategy A attempts to maximize the number of warm starts to reduce the overall system latency, while strategy B seeks to maximize the latency benefit. We can tell that even though strategy A results in $2\times$ more warm starts compared to strategy B, its overall latency is still 19.2% higher over strategy B. To illustrate with a simple example, if two functions have the same number of warm starts, but one has a container initialization latency of 100 ms and the other 1,000 ms, clearly warming up the function with 1,000 ms latency can reduce more system latency. The example indicates that 1) containerized functions have various characteristics, such as resource consumption, call frequency, and container initialization overhead, each impacting system latency differently; 2) although the number of cold starts reflects the performance of the system to some extent, it does not fully represent the overall system latency.

In this paper, we propose Incendio[1], a serverless framework with a priority-based scheduling method for reducing

---

[1]Incendio is a charm that conjures a jet of flames that could be used to set things alight in the Harry Potter fiction franchise.

latency incurred by cold start. Incendio is motivated by three key insights. First, functions vary significantly and should not be treated equally for the cold start. Different functions have various "priorities" in terms of latency benefits, which are determined by factors including the frequency of calls, resource usage, initialization overhead, etc. Second, traditional scheduling strategies, such as First-Come-First-Served (FCFS) or fair scheduling, fail to consider the priority of different functions and therefore result in suboptimal performance when scheduling the workloads. Third, executing requests directly whenever possible means that the container needs to be started and terminated at proper times so that functions can avoid the initialization and termination costs.

In light of these insights, we design a priority model based on Spearman's rank correlation coefficient. Further, we develop a hybrid Prophet-LightGBM prediction method to prewarm, terminate containers, and dynamically manage the container pool. Finally, we build a Clustered Reinforcement Learning (CRL) model with salient targeted scheduling policies for function priority. We note that existing public cloud vendors usually charge based on memory usage and actual usage time. Our method focuses on the response time of requests rather than changing the memory size, so pricing is not a concern or metric. Secondly, although the services provided by public cloud vendors only need to meet the Service Level Agreement (SLA) and there is no standard latency requirement in the SLA [20], lower response latency can attract more users and make more profits. In summary, this paper makes the following contributions:

- We find that reducing the number of cold starts has a limited impact on reducing the overall system latency (Section I). Thus, we investigate how cold start, function type, and scheduling strategy affect the performance of serverless computing. We observe that functions differ in priority in terms of latency benefits (Section II-B).
- We define function priority to indicate the importance of functions (Section IV), design a hybrid Prophet-LightGBM function arrival prediction method (Section V), and propose a Clustered Reinforcement Learning (CRL) based function scheduling strategy to minimize the latency cost in warming containers (Section VI).
- We design and implement Incendio on OpenWhisk and evaluate our framework with public traces. The experiment results show that Incendio speeds up the native system by $1.4\times$ and achieves 23% latency reduction compared with the state-of-the-art (Section VIII).

II. BACKGROUND AND MOTIVATION

A. Serverless Computing

Fig. 3 shows a typical function execution workflow in serverless computing, which is adopted by Openwhisk. The user passes in a customized function and invokes a request through the client (❶). The client downloads the container image according to the function (❷) and passes the request information to a controller (❸). Then, the controller verifies the user's identity and the validity of the request and dispatches the request to the actual execution environment by assigning a container
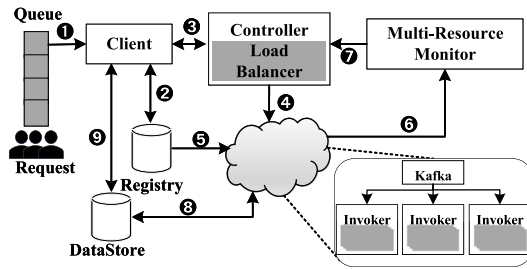
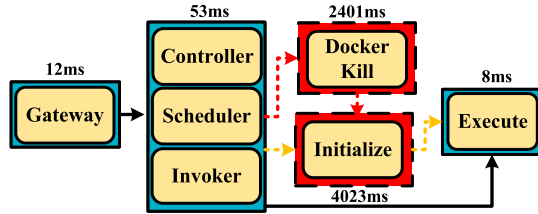Fig. 3.   Function execution workflow in serverless.

Fig. 4.   Execution latency breakdown of hello-world function. The number on the boxes indicates the latency of each step (ms). The dotted lines indicate unnecessary processes due to a cold start. The solid lines indicate the warm start process.

(a) CPU consumption.

(b) Memory consumption.

Fig. 5.   Resource consumption of eight functions.

through a load balancer (❹). The Runtime pulls the image from the Registry to start and initialize the container (❺). During the execution, a multi-resource monitor checks the resource usage (❻), which is sent to the controller for load balancing (❼). When the execution is completed, the results are stored in the database (❽). Then, the user can query the results through the client (❾). Inside the execution environment, Kafka works as a buffer storage request to avoid conflicts with high concurrency. The actual runtime happens in an invoker, each of which has a container pool to run the function.

The execution of a function requires the program (e.g., user code, language runtime, etc.) to be containerized inside the memory. However, it is expensive to keep all the resources required by functions in memory all the time, especially for the infrequent functions with short execution times. Meanwhile, functions differ in resource requirements and invocation frequencies. All the constraints and dynamics make it hard for the system to respond quickly.

B. Motivation and Case Studies

We conduct a series of case studies in a serverless system to demonstrate that cold start, function types, and scheduling policies significantly affect the system performance and application latency. We analyze the experimental results and further investigate the underlying causes. Specifically, we examine Openwhisk [4] using workloads from FunctionBench [21] and ServerlessBench [22] on a server equipped with a 4-core Intel i7-7700 CPU and 16GB Memory.

1) Why cold start matters? Fig. 4 shows the execution workflow of `Java-hello` function on OpenWhisk with the latency decomposition of each step. Compared to the time of initialization (4,023 ms) and termination
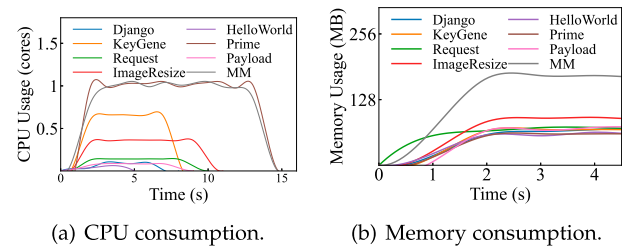
(2,401 ms), the execution time is almost negligible and only takes 8ms, which is just 0.2% of the initialization and 0.3% of the termination cost. Similar results can also be obtained on other serverless computing platforms.

2) Why function type matters? Confronted with the challenges of cold start, we aim to reduce overall latency for functions since few serverless providers specify latency as a standard in the SLA. When instantiating functions for invocation, existing platforms always adopt a fair policy, assuming that all serverless functions are equally important so that execution order and resource allocation need to ensure the completion of all tasks. However, the characteristics of functions determine the extent to which different functions will have an impact on the overall performance of the system. As shown in Fig. 2, the benefits of different functions vary greatly.

Further, we conduct experiments to demonstrate the intrinsic connection between function type and application latency. Functions differ in required resources (e.g., CPU, memory, and block I/O), call frequency, etc. All these factors have an impact on the latency of a function. Fig. 5 depicts the actual CPU (Fig. 5(a)) and Memory (Fig. 5(b)) resource consumption of eight functions. The differences in resource requirements indicate that functions need to be treated accordingly. By the same token, a function that is executed more frequently should have a different status from a function that is called less. Functions with long initialization should not be deployed with a function with quick starts. In a nutshell, to address the overall latency, function type should be considered in the design of a serverless system.

3) Why scheduling strategy matters? Scheduling is a critical factor in high-performance cloud systems [23], [24]. However, much less effort has been given to scheduling policies for serverless functions. Existing schedulers use techniques and policies that are ill-suited for the characteristics of serverless functions, i.e., function types and priorities, as we have discussed previously. For instance, OpenWhisk is one of the most widely used open-source serverless platforms. However, its default scheduler employs pure locality-based load balancing, co-locating invocations of the same function to as few invokers as possible. Another approach packs invocations to a randomly selected worker without accounting for the load, a technique shown to be unable to handle highly-skewed workloads [25]. Traditional policies like FCFS, though simple, cannot avoid the cold start or even incur unnecessary costs.
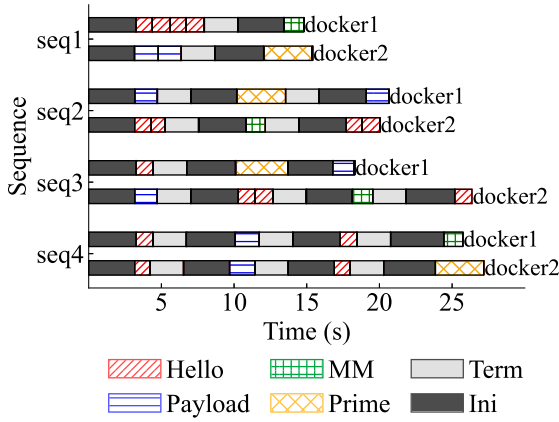
Fig. 6. Latency of different scheduling sequences (Y-axis). Hatched ones indicate execution time and no hatched ones represent initialization and termination time, respectively.

We conduct case studies to demonstrate the latency impact of different scheduling strategies. Consider that we have two docker containers to run 4× `Java-hello` functions, 2× `Payload` functions, 1× `Matrix Multiplication` function, and 1× `Prime-Number` function. Among them, `Java-hello` has the highest invocation frequency and the shortest execution time, followed by `Payload`, while `MM` and `Prime` have the lowest invocation frequency and the longest execution time. These functions arrive randomly to ensure generality. Fig. 6 shows the latency of different scheduling sequences. The FCFS policy (seq2), where all functions are scheduled in sequence, ensures fairness but does not guarantee container reuse, leading to numerous container initializations. If the Locality-Aware policy is used (seq3), when `Java-hello` and `Payload` are assigned to the same invoker, severe resource contention will occur, even worse than FCFS. Random scheduling (seq4) might lead to frequent container reuse for requests of the same function, but it could also result in new requests continually shutting down and reinitializing containers for previous requests, thereby leading to unstable performance. In addition, if long-running containers such as `MM` and `Prime` are prioritized, it will increase the waiting time for small functions, thereby affecting the average waiting latency of the entire system. The optimal scheduling strategy (seq1) prioritizes executing the most frequently called `Java-hello` functions with the lowest execution delays consecutively. It then schedules containers `MM` and `Prime` to corresponding invokers after completion. This not only avoids extensive container initialization but also reduces the waiting time for subsequent requests. The schedule minimizes the overall system latency by avoiding container initialization as much as possible.

However, this example is a simplified scenario, ignoring the dynamic nature of the workload and the variations in function execution time and initialization time. Therefore, it is necessary to devise a scheduling strategy that takes into account server resources, request arrival rates, and function characteristics at runtime.

III. DESIGN OVERVIEW

A. Incendio Architecture

Fig. 7 illustrates the architecture of Incendio. The core module is the **Controller**, responsible for requests receiving, scheduling, and managing the container pool. Specifically, the **Function Scheduler** orchestrates the deployment of requests. It receives output from the **CRL Model** and invokes the Deploy operation to execute requests. The Deploy operation deals with container reuse and creation. If an idle container corresponding to a request exists in the pool, it can be reused (warm start); otherwise, a new container is created for the request under sufficient resource conditions (cold start). The **CRL Model** is tasked with making scheduling decisions based on current requests and the state of the container pool. It takes resource information from the **Function Info Table** and priority data from the **Priority Model** as inputs. New scheduling decisions are made after requests on the server are completed and are outputted to the **Function Scheduler**. The **Priority Model** updates the priorities of functions. It processes resource usage information from the **Function Info Table**, along with data on the execution and termination latency of function containers. This model recalculates and updates priorities after container termination. These updates are sent to both the **CRL Model** and the **Function Info Table**. The **Function Arrival Predictor** forecasts the arrival of requests. It utilizes historical arrival data of functions from the **Function Info Table** to predict the number of function calls in the next minute. These predictions are then conveyed to the **Container Manager**. The **Container Manager** oversees the container pool, performing Update operations, including pre-warming containers and timely termination. The **Resource Monitor** tracks the status and resource usage of the container pool, reporting these metrics to the **Function Info Table**. It updates the container's initialization overhead and live status immediately after deployment, the resource usage information after execution, and the remaining information after container termination. Finally, the **Function Info Table** is a data structure that stores critical information related to function execution and calls. It includes the priority of functions, resource usage, and various latency metrics (response, execution, termination). Note that the table stores the priority of a category of functions, not individual requests. If the table is not updated, all requests from the same function retain the same priority level.

The workflow of Incedio, encompassing request scheduling and container pool management, is presented as follows. For request scheduling, when a request is authenticated, authorized, and certified(❶), it is passed to the **Controller**(❷). The request is assigned a priority based on the **Function Info Table**. Until this table is updated, all requests for the same function share the same priority. When a request is completed, the **CRL Model** outputs a new scheduling decision to select the request for execution (❸). The **Function Scheduler** invokes Deploy to execute the selected request(❹). For container pool management, the **Function Arrival Predictor** forecasts the number of function calls in the next minute based on historical arrivals of the function and sends the information to the **Container Manager**(❺). This module manages the lifecycle of containers through update
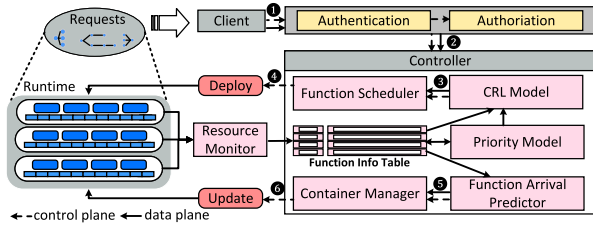
Fig. 7. Incendio architecture. The pink boxes are the new modules we added. The yellow ones are original modules.

operations, prewarming or terminating containers based on the anticipated number of future requests(❻). Request scheduling and container management are not executed synchronously but rather operate independently in cooperation with each other, built upon the operations of container creation, reuse, and termination.

B. Challenges and Key Design Choices

Challenge 1: How to define function priority? The functions are of various types. Their demands on resources, distribution of arrivals, etc., are also significantly different and highly dynamic. Therefore, how to consider the characteristics and properties of the function to define its priority is a key question for the serverless system.
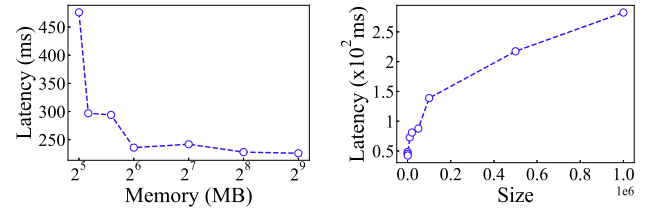
Design 1: Function Priority Definition based on Spearman's Rank Correlation Coefficient (Section IV). To define the function priority, we use Spearman's rank correlation coefficient to explore the effects of various attributes and factors on latency separately. Parameters that have a positive or negative correlation on the performance are selected based on the value of the correlation coefficient.

Challenge 2: How to prewarm containers? The function calls are volatile since their frequency and pattern vary significantly in different applications. Confronted with the dynamic of functions and unpredictable user behavior, it is difficult to prewarm containers efficiently and precisely.

Design 2: Hybrid Prophet-LightGBM Request Prediction Model and Dynamic Block Management (Section V). To adjust the dynamic environment promptly, we use hybrid Prophet-LightGBM, a model that aggregates historical data while taking into account trends, seasonality, and holidays, to predict the arrival distribution of functions. We design a dynamic block adjustment scheme to complement the prediction model, which extends the serverless computing framework and timely deals with function prediction.

Challenge 3: How to schedule functions facing runtime dynamics? The traditional scheduling strategy adopts a one-size-fits-all fixed policy for all requests, which conflicts with the concept of function priority. The question is how to perform gradient scheduling with function priority difference.

Design 3: CRL based Scheduling Policy (Section VI). With the function priority defined, a CRL approach is designed to make decisions on the scheduling choices of functions to maximize the overall performance. A table with function priorities is maintained and updated in real-time.

(a) Memory-Latency (Matrix). (b) InputSize-Latency (Payload).
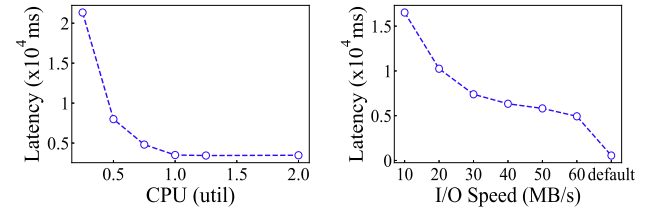
(c) CPU-Latency (Prime number). (d) BlockIO-Latency (Disk R/W).

Fig. 8. Spearman's rank correlation coefficient between different factors and function latency.

IV. FUNCTION PRIORITY DEFINITION

In this section, we define function priority by adopting the statistical approach of Spearman's rank correlation coefficient. Spearman's rank correlation coefficient assesses how well the relationship between two variables can be described. It takes values in the range of [-1,1], indicating either perfect positive or inverse correlation. Compared with the Pearson correlation coefficient or machine learning models, it does not concern whether the dataset is linearly related but only focuses on the monotonic correlation of the dataset, which has better generality and simplicity. In general, two variables are considered to be strongly correlated when the absolute value is greater than or equal to 0.8. As shown in Fig. 8, we evaluate and get the trend in function latency as CPU resources, memory resources, block I/O speed, or input size increase, respectively. We observe that the latency elevates when the resource allocation increases (Fig. 8(a), 8(c), 8(d)) or input size decreases (Fig. 8(b)). Take CPU latency as an example to calculate the correlation absolute value. Since the upper limits of CPU allocation are set to [0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.74, 2] and the corresponding latency (ms) are [21334, 7995, 4811, 3505, 3441, 3475]. We first rank the data, resulting in arrays for CPU [1, 2, 3, 4, 5, 6, 7, 8] and latency [8, 7, 6, 5, 3, 4, 1, 2]. Utilizing the spearmanr(data1, data1) function provided by the scipy.stats module in Python, we input the ranked data to compute the coefficient, obtaining a value of -0.952. The correlation absolute value between the function latency and factors are 0.96 (Memory), 0.95 (CPU), 1 (Block I/O), and 0.93 (Input Size), respectively, which shows strong dependences and therefore, should be taken into account for the priority model. It should be noted that latency does not indefinitely increase (decrease) in proportion to the resource allocation (reduction). Consider a simple 'Hello World' program, once the allocated memory surpasses a certain threshold, its latency tends to stabilize.

Meanwhile, to validate the universality of our method, we perform normalization and discretization on the same dataset

TABLE I
PRIORITY-RELATED PARAMETERS

Parameter	Description
CPU	CPU allocation
Mem	Memory allocation
I/O	I/O allocation
Size	Input size of the parameter
Init	Time from invoke time to start
Exec	Time from start to end
Term	Time from termination command to success
Freq	Function invoked frequency
Clock	The interval time between the last arrival

and then repeat the experiment using the Kendall correlation coefficient, Chi-Squared test, and decision tree methods. The results are consistent with Spearman's rank correlation coefficient, demonstrating its effectiveness.

In addition to the resource metric, the initialization, execution, and termination time also have huge impacts on the overall latency. Thus, we add these factors to the composition of the priority model. We also consider the frequency and the most recent call of functions to the latency, and Table I shows parameters in our priority model.

All resource usage values have been normalized and standardized. Taking CPU and Memory as examples, considering the configurations provided by typical public cloud vendors, the number of container memory and CPU cores used is proportionally increased. For instance, Google Cloud Function 2nd gen [26] provides 0.583 vCPUs with 1GB of memory, and when the memory is increased to 2GB, it provides 2 vCPUs. Therefore, we adopted a 2:1 Memory-to-CPU ratio, where every 512MB of memory size corresponds to 1 CPU core. For example, a 128MB memory usage corresponds to a value of 0.25, and a 0.2 core CPU usage corresponds to a value of 0.2. We denote the function priority P as $F(\cdot)$:

$$P = F(CPU, Mem, I/O, Size, Init, \\ Exec, Term, Freq, Clock) \quad (1)$$

which is a general model to accommodate the dynamic nature of serverless computing. To meet the diverse scenarios in a real environment, we also design two models in practice: the Linear model and the Loglinear model. Each of the models has its advantages to match corresponding conditions. For the Linear model, the definition is as follows:

$$p_i = \sum_{i=0}^{8} w_i \cdot X_i + \epsilon, 0 \le w_i \le 1 \quad (2)$$

where X_i represents a factor of the priority function and w_i represents its weight value, satisfying $\sum w_i = 1$, and ϵ is a general constant. For different types of functions, the weight of each factor varies and can be trained from historical datasets. For Loglinear model, its general form is defined as follows:

$$p_i = \frac{\prod_{i=0}^{k} w_i \cdot X_i}{\prod_{i=k+1}^{8} w_i \cdot X_i}, 0 \le w_i \le 1, 0 \le k \le 7 \quad (3)$$

where X_i represents a factor of the priority model and w_i is its corresponding weight value. A factor can either be positive or inverse, and we use k to represent the number of positive variables. For example, the more frequently the function is called, the more important it is likely to be. Hence we set Freq to be positive on the top of Equation (3) and give it an appropriate weight value to indicate the degree of a positive relationship. We simplify the equation logarithmically to obtain the loglinear model.

The linear model is relatively simple but has good universality and can adapt to general scenarios. In comparison, the loglinear model can be better tuned to scenarios where the load has certain characteristics through parameter tuning. For instance, one certain type of function dominates the workload. Although the loglinear model has the above advantages, extreme cases might occur when scenarios transform or parameters are inappropriately chosen. In Section VIII, we will evaluate and analyze the results of these two models.

V. HYBRID PROPHET-LIGHTGBM FUNCTION ARRIVAL PREDICTION

When the function priorities are defined, for those requests for high-priority functions, we should avoid the cold start as much as possible. Therefore, we introduce a time series prediction model to predict the function arrival distribution and manage the live container pool. We prewarm containers in advance to avoid initialization if the cost is high or terminate containers when there are no future calls over a period of time.

A. Why Hybrid Prophet-LightGBM Model?

To better pre-process high-priority functions that are likely called in the future, we need to predict the distribution of function arrivals. Classical time series prediction models such as Exponential Smoothing(ES) and ARIMA typically require manual configuration of the model and uncertainty parameters. Although machine learning-based models such as long short-term memory (LSTM) can capture long-term dependencies in the data, they require huge training samples and a large amount of training time. It cannot easily incorporate non-time-based external features or noise and uncertainty into its predictions. While LightGBM is known for its speed and high accuracy, it is sensitive to feature selection and emphasizes specialized statistical knowledge and experience. The latest Prophet model combines time series decomposition and machine learning fitting for prediction. However, its accuracy in predicting complex patterns is compromised and long-term prediction is unreliable.

To overcome these problems and achieve generalizable and scalable prediction, we build a hybrid Prophet-LightGBM prediction model. The advantage of our model is twofold. Firstly, this method can leverage the seasonality, trend, and confidence interval provided by Prophet to extract more meaningful features for LightGBM and improve prediction accuracy. Secondly, as LightGBM has fast gradient descent and low memory usage, combined with the Prophet-specified special event settings and bound limits, models can avoid overfitting or iteration traps.

B. Dynamic Environmental Function Prediction

To begin with, we preprocess the data using Prophet according to the specific characteristics of the serverless functions. The general model of the Prophet is shown as follows:

$$y_t = g_t + s_t + h_t + \epsilon_t \tag{4}$$

where g_t is the trend function, s_t represents the value of the acyclic variation, and h_t represents the cyclic variation and the effect of irregular holidays. The error term ϵ_t represents any particular variation that the model cannot accommodate and is assumed to conform to a normal distribution. Due to the limited degree of parallelism for functions in serverless computing, we utilized a logistic regression function with a maximum parallelism limit and dynamically adjusted g_t according to the constraints. To account for the seasonality trends s_t in serverless functions, we also employed the Fourier series to flexibly adjust for seasonal changes on a daily, weekly, monthly, and yearly basis. Additionally, we took into account the impact of special events h_t such as "Black Friday" and holidays like "Thanksgiving". Different holidays were treated as independent models, and different window values were set for each holiday.

According to the results from Prophet, we can derive the trend, seasonal, and special event components, and use them as features for LightGBM. Through experiments, we have found that these features are more reasonable and effective compared to features such as mean, median, max, min, and standard deviation. However, tuning hyperparameters for LightGBM often requires expert knowledge and extensive trial and error. To address this issue, we utilized the state-of-the-art Optuna tool [27] and pruned the search space using the upper and lower bounds provided by Prophet, which further reduced the tuning time. After pre-training for a period of time, we selected an L1 regularized regression model as the target, which can achieve prediction within minutes with fewer than 200 iterations, meeting our requirements for practical use.

Based on the prediction model, we predict the arrival of functions with different priorities and manage function runtime dynamically. For functions with upcoming requests, the manager will update the container pool at regular intervals. If there is an idle container that corresponds to an upcoming request and the termination cost is greater than the time delay between now and the upcoming, the container will be kept alive. Otherwise, the container will be terminated to release resources. If a request is predicted to come but there lack of live containers, the manager will prewarm the container if there are sufficient resources.

VI. THE CLUSTERED REINFORCEMENT LEARNING BASED SCHEDULING POLICY

In the prior section, we leverage the hybrid Prophet-LightGBM model to manage the container pool. When the container status is determined, we need to schedule functions based on the arriving function requests and corresponding function priorities. Here, we thoroughly analyze the problem and build a Clustered Reinforcement Learning (CRL) method to make the optimal scheduling choice.

A. Problem Formulation

Consider that K function requests with N different types arrive at time t. These different functions are represented by the set $\mathcal{F}$, where $\mathcal{F} = [F_1, F_2, \cdots, F_N]$. Each element represents a different kind of function. For F_i, its jth request is denoted as $r_{i,j}$. $\mathbf{J}_i = [r_{i,j}]$ represents the request set of F_i that satisfies $\sum_{i \in \mathbf{N}} \sum_{j \in \mathbf{J}_i} r_{i,j} = K$ and $\mathbf{J}$ represents the set of all $r_{i,j}$. For a given server with memory M and CPU cores V, the scheduling of a function at time t is treated as a variable:

$$\mathbb{I}u_{i,j} = \begin{cases} 1, & j\text{th request of function } i \text{ is selected} \\ 0, & j\text{th request of function } i \text{ is not selected} \end{cases} \tag{5}$$

In addition, the execution time and resource consumption of each function should satisfy the following constraints.

$$\begin{cases} t_{i,j} \cdot \mathbb{I}u_{i,j} \leq T, \forall i \in N, \forall j \in \mathbf{J}_i \\ \sum_{i \in N} \sum_{j \in \mathbf{J}_i} m_j \cdot \mathbb{I}u_{i,j} \leq M, \\ \sum_{i \in N} \sum_{j \in \mathbf{J}_i} v_j \cdot \mathbb{I}u_{i,j} \leq V \end{cases} \tag{6}$$

where $t_{i,j}$ represents the response latency of $r_{i,j}$ and T represents the timeout limit for the request. v_j represents the CPU resource consumption of task j, the result of calculating the container configuration CPU_quota/CPU_period. m_j represents allocated memory. V and M refer to the currently available CPU and memory consumption. We do not specify the restrictions of Block I/O because Block I/O can change the relative priority by adjusting the $blkioweight$, and thus no special restrictions are needed.

Based on the definitions above, we formally define the Function Scheduling Priority Problem (**FPS Problem**) in serverless computing as below:

$$\max_{\boldsymbol{u}} \sum_{i \in N} \sum_{j \in \mathbf{J}_i} \mathcal{I}_{i,j} \cdot \mathbb{I}u_{i,j} \tag{7}$$

where $u = [u_{i,j}, t]$ represents the matrix of scheduling selection and $I_{i,j}$ represents the priority of $r_{i,j}$. Since the priority of each request is equivalent to the priority of its corresponding function, all requests associated with the same function share the same priority within the period before the function's priority is updated.

B. Problem Analysis

The FPS problem is, in fact, a 0-1 multiple backpack problem. It can be seen as N function requests (items) of different resource requirements (weight) and priority (value) being scheduled (packed) under available resource limits (available weight) to maximize the weighted priorities (total value). Hence, the FPS problem is equivalent to the backpack problem and is also an NP-complete problem.

*1) **Why Reinforcement Learning?*** Existing heuristics like Dynamic Programming (DP), and Branch and bound (BB) are either time-consuming or inaccurate, failing to satisfy serverless scenarios that require fast responses with low cost and high accuracy. We have conducted an experiment on DP to demonstrate the drawback. When the load is low, DP is fast (around tens of milliseconds). However, its latency reaches even tens of seconds under high load, completely blocking function scheduling, let alone accuracy.
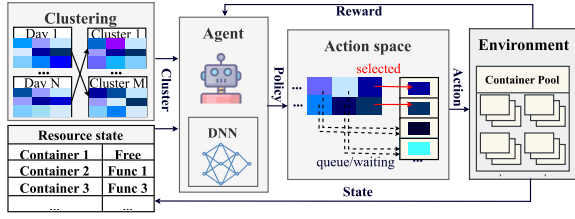
Fig. 9. The workflow of the CRL algorithm.

Reinforcement Learning (RL) has been widely suggested to solve such NP-complete problems effectively [28], [29]. In general, RL works like this: at each decision epoch, the agent will make a decision based on the current state of the environment. Once the decision is made, a reward would be provided to the agent and the state of the environment would be updated for making future decisions. The agent tries to maximize the cumulative rewards over time. With RL, the FPS problem can be tackled in a Markov Decision Process (MDP), which is a five-tuple $\langle S, A, P, r, \lambda \rangle$, where S denotes the set of states; A denotes the set of actions; P denotes the transition probability distribution; r denotes the reward function and $\lambda \in [0, 1]$ denotes the discount factor for future rewards. Note that different optimization problems have quite different objectives, constraints, and variables. To tackle the FPS problem, different components of RL need to be designed specifically.

2) Why Clustered Reinforcement Learning (CRL)? In serverless computing, function codes and request arrival distribution are dynamic, while existing RL approaches usually assume a fixed environment. Hence, RL should not be directly applied in our scenario. In this regard, direct use of RL models will lead to poor decision performance. Furthermore, it takes a long time to train a traditional RL model, which might be a bottleneck when scaling up. Accordingly, to solve the FPS problem, we need to learn the current environment. One key idea is that the more similar historical moments, the more similar the environment is. Such an idea is the core of clustering. Inspired by this idea, we divide the historical data into different clusters for learning, which adapts to the dynamically changing environment and accelerates the learning process. CRL is more lightweight, effective, and practical compared to traditional RL.

C. Clustered Reinforcement Learning for Scheduling

We introduce the key designs of CRL approach for function scheduling, i.e., the environment modeling, state space, action space, reward function, and optimization, which should be specified according to the FPS problem.

Environment. By analyzing the data from the historical data, we build an environment dataset, the Historical Environment Lambda. We define the historical environment $\mathcal{E}$ as the set of environments. We have $\mathcal{E} = [e_1, e_2, \cdots, e_{N'}]$. Through environment definition, we can find a similar environment e by clustering algorithms, i.e. k Nearest Neighbors (kNN), where $e = kNN(E, Z)$. Z denotes the sensing data. For the RL predictor, the environment e can be seen as a matrix form. More specifically, for matrix e, one of the dimensions can come

to represent all the function types reached at the moment t, and the other dimension represents the number of reaches. Thus the normalized matrix form of the environment e is $e = [N_j]_{1 \times N}, \forall I_j, V_p \in \mathbb{R}$. It represents the distribution of arrivals of functions.

State. We normalize the representation of the state, which is the current task selection of the system. Specifically, the state is defined by a matrix S, where the elements of each position can be either 0 or n, where n is not equal to 0. Note that n represents the number of times each function has been selected, but n cannot exceed the total number of current arrivals of that function, otherwise, it is not selected and is expressed as $\mathcal{S} = [s_i]_{1 \times N}, \forall s_i \in [0, N]$.

Action Space. At each time point, the scheduler selects the function that it wants to schedule. It is assumed that the j function is selected for scheduling. If a j-function corresponding to a capacitated function in the busy state exists, the options to be scheduled include starting a new container to execute or waiting and reusing the corresponding container. If it does not exist, only the first behavior is available. For both cases, the prerequisite for starting a new container needs to satisfy the resource constraints defined in Equation (6).

Reward. Reward is an important indicator of the effectiveness of a learning model. We carefully design the reward signal to guide the agent toward the desired solution. The ultimate goal of our CRL approach is to minimize the weighted average latency of the task. Thus, the reward is

$$r(t) = \begin{cases} \sum_{i \in N} \sum_{j \in \mathbf{J}_i} I_{i,j}, & \text{terminal state} \\ 0, & \text{otherwise} \end{cases} \quad (8)$$

Specifically, the reward value is $\sum_{i \in N} \sum_{j \in \mathbf{J}_i} I_{i,j}$ when the system enters the final state, i.e., when all requests in the current system are completed, and 0 otherwise.

Optimization. With the above key elements, we leverage Deep Q-learning $Q(s, a; \theta, \mathbf{J})$ [30], where θ denotes the adjustable parameter vector of neural networks. It estimates the value of executing an action from a given state s. Formally, given the feature space X which consist of the environment e and the initial state s_0, we have

$$\mathbf{u} \leftarrow \mathcal{F}_1(\mathbf{J}, \mathcal{X}) = \mathcal{F}_1(\mathbf{J}, (e, s_0)) = Q(s, a; \theta, \mathbf{J}) \quad (9)$$

Based on the design, the CRL approach is presented in Algorithm 1. We utilize Deep Q-Network for training, employing a four-layer fully connected neural network with dimensions 1024, 512, 512, and 256, respectively. In the forward method, the input state passes through the first layer, followed by the application of the ReLU activation function. After proceeding through the second layer and followed by another ReLU, the data flows through the third layer and a subsequent ReLU before reaching the output layer. The output layer generates the Q-values for each action, representing the expected value of taking each possible action under a given state. We employed the Adam optimizer for optimizing network parameters and used mean squared error as the loss function. In the agent, we set a discount factor and initialized the exploration rate (a part of the ϵ-greedy strategy to balance exploration and exploitation). We also establish the minimum value and decay rate for the

Algorithm 1 Clustered Reinforcement Learning (CRL)

Input: $\mathcal{E} \leftarrow$ historical Environment, $s_0 \leftarrow$ initial state, $Z \leftarrow$ current configuration, $e \leftarrow$ Clustering($\mathcal{E}, Z$)

1: **function** TRAINING()
2:     **while** not reach the terminal state s_N **do**
3:         $\mathcal{L}(s, a \mid \theta) \leftarrow (r + \max_a Q(s', a \mid \theta) - Q(s, a \mid \theta))^2$ ▷ Update DNN parameters
4:     **end while**
5:     $\theta^* \leftarrow \arg\min \mathcal{L}(s, a \mid \theta)$ ▷ Obtain optimal parameter
6:     **return** (e, s_0, θ^*)
7: **end function**
8:
9: **function** PREDICTION()
10:     $Initialization\,(e, s_0, \theta^*)$
11:     $\mathbf{u} \leftarrow \widetilde{\mathcal{F}}_1\,((e, s_0)\,; \theta^*)$ ▷ make prediction
12:     **return** $\mathbf{u}$
13: **end function**

---

exploration rate, as well as the batch size. Two networks are created: a policy network for action selection and a target network for calculating target Q-values. Actions are selected randomly using the ϵ-greedy strategy, or by choosing the action with the highest expected Q-value.

Convergence Analysis. The Deep Q-learning technique has been proven to gradually converge to the optimal policy under a stationary MDP environment and sufficiently small learning rate. Hence, the proposed CRL predictor will converge to the optimal policy when 1) the environment evolves as a stationary, memory-less Semi-Markov Decision Process and 2) the DNN is sufficiently accurate to return the action associated with the optimal $Q(s, a)$ estimate.

D. Fairness Guarantees With Service Level Objective

Although the priority-based CRL scheduling strategy can ensure that important functions are scheduled preferentially, it may result in lower-priority functions being starved and not executed for extended periods. To avoid such unfairness, we have established an SLO standard to target response latency and prioritize scheduling those requests whose waiting time is approaching the SLO limit. Subsequently, we use the CRL model to schedule the remaining functions. We want to emphasize that the SLO refers to the response latency of a request, which is the time elapsed from the invocation to its scheduling, excluding the execution time. Specifically, we provide users with an SLO determined by default and adaptive options. The default SLO is a fixed value applicable to all functions, while the adaptive SLO is calculated as the product of the initial startup latency (cold start) of a function and a fixed percentage. The SLO standard is determined as $\max(SLO_{default}, SLO_{adaptive})$. Consider a function A with a cold start latency of 2,000 ms, its SLO would be max(400, 30% * 2000) = 600 ms. In our system design and evaluation section, we set the default SLO to 500 ms and the adaptive percentage to 30%. This approach maximizes fairness among requests, preventing low-priority requests from experiencing extended waiting times, while also ensuring that high-priority requests are scheduled preferentially.

VII. IMPLEMENTATION

Simulator. We have developed a serverless core workflow simulator, which includes a function scheduler, a container pool manager, and a request queue. The function scheduler implements various scheduling algorithms such as Random, Locality-aware, and FCFS, while the container pool manager provides basic functionalities including container creation, deletion, and reuse.

Openwhisk Modification. We have implemented Incendio on OpenWhisk, including adjustments to the Controller, the addition of the Function Info Table, and the Priority Model. Whenever a request arrives, the Function Info Table communicates with the Client and the Controller to update request data and supplement it by querying the CouchDB. The Priority Model calculates and updates the function priority based on the Function Info Table data. We added a Predictor submodule to the Controller, which predicts function invocation based on the historical data obtained from the Function Info Table. According to the prediction results, the Load Balancer terminates and creates containers at regular intervals. In addition, we have replaced the Load Balancer's scheduling logic with the Scheduler, which maintains a request queue and selects requests upon the current container pool status and function priority using the CRL model. The selected requests are then sent to the Kafka message queue.

VIII. EVALUATION

In this section, we evaluate the performance of Incendio, intending to answer the following questions.

- How does Incendio perform compared to the existing serverless platforms and approaches? (Section VIII-B)
- To what extent can internal components contribute to the overall performance? (Section VIII-C)
- How does the priority model improve the performance with different forms and parameters? (Section VIII-D)
- How do the hybrid Prophet-LightGBM function predictor and container management strategy affect the performance? (Section VIII-E)
- Can SLO effectively ensure scheduling fairness among requests? (Section VIII-F)
- What is the cost of the CRL-based scheduling strategy compared to other heuristics? (Section VIII-G)

A. Methodology

Testbed Settings. We deploy Incendio to a dedicated local cluster with 16, 8-core servers using Intel i7-7700 CPU with 64GB Memory each. The OS is Ubuntu 16.04 with Linux kernel 4.15.0. Each server is connected to a 1Gbps switch over 1Gbe NICs. We use one of the 40-core servers to host the controller, API gateway, CouchDB, and other system components. The remaining servers each host an invoker along with a container pool to run functions using Docker.

Workloads and Simulation. We used Azure public trace 2019 [10] to simulate a real serverless computing environment. The raw trace provides data from the application layer. Since the

application consists of multiple functions, we consider each function to be of equal status and allocate the memory equally for ease of processing. Some data in the trace can interfere with the experiment. For example, if the number of calls is very small, container reusing will disappear. Therefore, we will exclude applications with less than five function calls. To prevent overfitting, we supplement our experiments with Azure trace 2021 [12], Alibaba trace 2021 [31], and Huawei trace 2023 [32] in validating the general performance.

To consider the impact of real functions, we also adopt the following loads from FunctionBench [21] and ServerlessBench [22], including `Matrix Multiplication`, `Prime Number`, `Request`, `Floating Point` and `Cifar`. We assign 128M, 64M, 32M memory, and $1.0\times$, $0.75\times$, $0.5\times$ CPU core usage limits randomly to the workloads. The arrivals of function requests satisfy the Poisson process, where the inter-arrival time follows an exponential distribution.

Baselines. We compare Incendio against two state-of-the-art techniques: HotC [9] and FaasCache [14], as well as native serverless platform OpenWhisk [4]. The mechanisms of the techniques are given as follows:

- **HotC**: it maintains a live container runtime pool and adaptively updates the pool over time. HotC uses exponential smoothing to predict function arrival and the Markov chain method to allocate resources.
- **FaasCache**: different from traditional caching, it uses Greedy Dual's cache-evict policy to control the live container pool with different priorities.
- **Native**: when requests arrive, it creates a new container or reuses a live one if possible. OpenWhisk removes containers with the least recently used (LRU) policy and keeps a container live with a time-to-live (TTL) of 10 minutes.

B. The Overall Performance

Improvement on the Overall Latency. We first evaluate the overall latency improvement of the entire system by measuring the latency speedup across real-world serverless workflows. We also compare Incendio's model with the three baselines. We use the same training dataset for all systems and evaluate performance on a separate test dataset under four different workload traces.

Table II shows the overall speedup and component analysis. Incendio significantly outperforms all other alternatives, achieving the highest acceleration ratio of 14.8%, the lowest of 6.1%, and an average of 10.4% compared to the second-best model, FaasCache. Our proposed CRL model for scheduling and Prophet-LightGBM model for pool management fully capture the dynamic invocation pattern. HotC performs the worst, and the average result is 17.9% lower than Incendio. The main reason is that HotC does not consider the priority of different functions.

Improvement on Cold Start & Functions with Priority. In addition to latency speedup, we also analyze invocation details with five functions using the Azure Trace 2019. Fig. 10 shows the breakdown of different invocations. We can observe that the native platform drops 55.9% of total requests due to its cold

TABLE II
THE OVERALL SPEEDUP DUE TO DIFFERENT METHODS

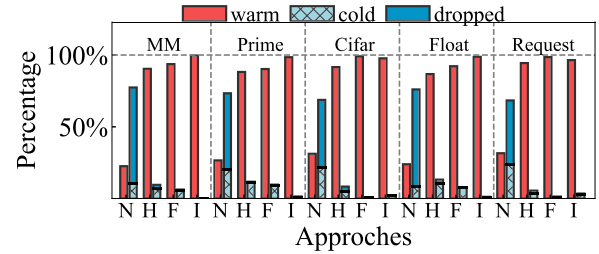| Methods	Native	HotC	FaasCache	Incendio	I-wo-S	I-wo-P
Azure19	1.00	1.13	1.21	1.39	1.33	1.24
Azure21	1.00	1.15	1.23	1.37	1.31	1.29
Alibaba	1.00	1.06	1.12	1.19	1.17	1.16
Huawei	1.00	1.22	1.30	1.43	1.37	1.34

Fig. 10. State distribution of the five requests under four methods. "N" denotes Native, "H" denotes HotC, "F" denotes FaasCache and "I" denotes Incendio.
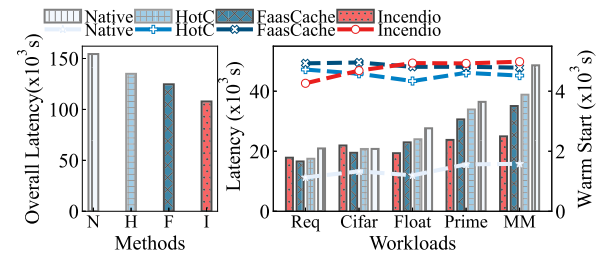

Fig. 11. Response latency and the number of warm starts. In the left figure, bars represent the overall latency of each method. The right bars represent latency distribution and the lines show the number of warm starts.

start overhead, followed by HotC (2.3%), FaasCache (0.3%), and Incendio (0.6%). This phenomenon has also reflected that cold start results in not only an increase in latency but also a decrease in system load. In other words, Incendio allows more functions to be executed under the same configuration compared to the native system, boosting the system load capacity. We also find that 27.2% invocations of the native system are 'warm start', which is significantly lower than HotC (91.08%), FaasCache (97.16%), and Incendio (95.17%). In terms of warm start, FaasCache has a slight advantage over Incendio(1.99%). However, the number and ratio of cold starts or warm starts cannot exactly reflect the actual system performance. As shown in Fig. 11, though FaasCache outperforms in the number of warm starts, Incendio's overall latency is the lowest. Compared to the Native, HotC, and FaasCache, Incendio is 30.3%, 20.1%, and 13.5% lower than their latencies, respectively. For various serverless functions, although Incendio has a slight lag in `Req` and `Cifar`, it outperforms functions of higher priorities (i.e., `Matrix`, `Float`) in terms of latency benefit. Table III shows the total latency benefit: 34.2 (Native), 52.5 (HotC), 55.3 (FaasCache), and 59.9 (Incendio). The result is consistent with the discussion in Section I. This observation also proves that latency

TABLE III
RESULTS OF THE OVERALL LATENCY BENEFITS ($10^3 s$)

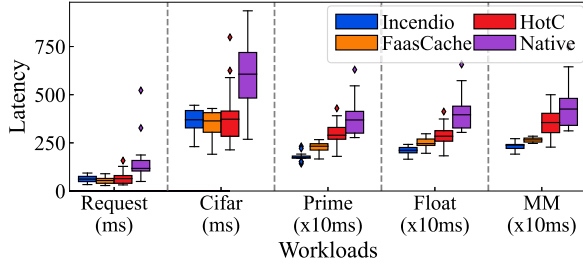| Methods	Native	HotC	FaasCache	Incendio	I-wo-S	I-wo-P
Results	34.215	52.498	55.307	59.881	58.280	57.934

Fig. 12. Overall tail latency of four methods. The top horizontal line represents the 99th percentile latency, while the box from top to bottom shows the 25th percentile latency, 50th percentile latency, and 75th percentile latency.

benefit varies with different functions and the number of warm or cold starts is not strictly equivalent to latency reduction.

Improvement on the Tail Latency. Since tail latency is a key metric in serverless function execution. We further use the function invocation traces to emulate workflow invocation patterns. The trace is a record of function invocations, including their quantity and time.

Fig. 12 presents the results of the five workloads mentioned above. Overall, concerning tail latency across three different workloads, MM, Float, and Prime, Incendio performs the best, achieving 9.3%, 14.5%, and 20.1% average latency improvement respectively over the method with the second-lowest latency among the three workloads. This is attributed to the high priority and frequency of these workloads, allowing for the full utilization of scheduling and pre-warming benefits. Regarding Cifar and Request workload, Incendio, Faas-Cache, and HotC performed almost identically. Incendio significantly reduces the latency of the majority of functions and also shows good stability in tail latency. Although it may temporarily sacrifice some functions due to their priority, the system capacity is improved, which means these functions can meet the user's latency requirements and could be completely accepted compared to the cold-start overhead of the original system. This trade-off is deemed significant from the provider's perspective.

C. Analysis of Components

We evaluate Incendio's internal components and analyze their contributions to the overall performance. We implement two breakdown versions of Incendio: (1) **Incendio-wo-S** has a predictor model and the corresponding container pool management but it cannot scheduler function with CRL. (2) **Incendio-wo-P** has a CRL-based scheduler while using the traditional LRU eviction policy and fixed keep-alive policy to manage the container pool.

Table II shows the comparison results in latency speedup. We can see that Incendio-wo-S outperforms Incendio-wo-P in most
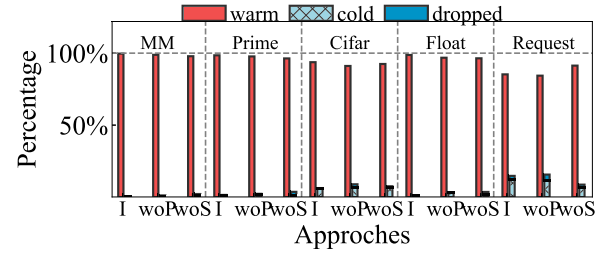

Fig. 13. State distribution of the five requests under component methods. Here "I" denotes Incendio, "woP" denotes Incendio-wo-P, "woS" denotes Incendid-wo-S.
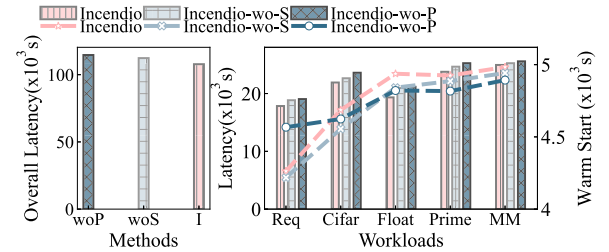

Fig. 14. Response latency and the number of warm starts. In the left figure, bars represent the total latency of each component. On the right, bars represent detailed latency distribution and lines show the number of warm stats.

cases. This is because predictor model can create appropriate containers in advance, which reduces the number of containers that need to be scheduled to some extent and ensures that there are idle containers available for direct reuse every time a request arrives.

Fig. 13 shows the invocation details for each component in great depth under Azure Trace 2019. Because all components are prioritized, the proportions of warm start, cold start, and dropped requests are practically the same. Due to the low priority of function `Request`, Incendio and Incendio-wo-P have a higher number of cold starts. The predictor makes the most benefit by correctly anticipating future calls. We then investigate the overall latency shown in Fig. 14. Incedio outperforms the others with a narrow margin which is also consistent with the result in Table III under four traces. As the overall results depict, Incendio has the highest total benefit, followed by Incendio-wo-S and then Incendio-wo-P. The reason why Incendio-wo-S slightly outperforms Incendio-wo-P is that the scheduler module only considers the most current calls, ignoring future requests. However, both Incendio-wo-P and Incendio-wo-S are lower than Incendio, illustrating the necessity to combine scheduling and predicting.

Fig. 15 shows the tail latency under Azure trace 2019. Since Predictor is somehow taking up part of Scheduler's work, i.e., Predictor creates free containers in advance for direct reuse and terminates unnecessary containers in advance to reserve resources, Incendio-wo-S performs better.

In all, no "module" is an island. In the face of the dynamic serverless environment, only the collaboration of two modules can achieve better performance.
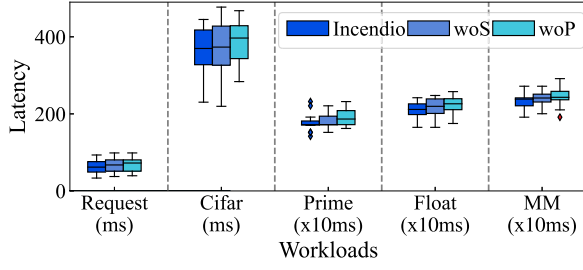
Fig. 15. Sensitive analysis of tail latency. The top horizontal line represents the 99th percentile latency, while the box from top to bottom shows the 25th percentile latency, 50th percentile latency, and 75th percentile latency.

TABLE IV
PRIORITY MODEL DETAILS. THE VALUES IN LINEAR REPRESENT THE
WEIGHTS OF THE CORRESPONDING PARAMETERS, AND THE SYMBOLS IN
LOGLINEAR REPRESENT THE CORRELATION BETWEEN THE
CORRESPONDING PARAMETERS

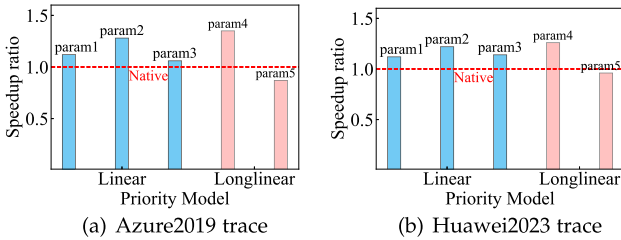| Linear	CPU	Memory	I/O	Init	Exec	Term	Freq	Clock
param1	0.3	0.3	0	0.1	0.1	0.1	0.05	0.05
param2	0.1	0.1	0.1	0.2	0.2	0.2	0.05	0.05
param3	0	0	0	0.2	0.2	0.2	0.2	0.2
Loglinear(log)	**CPU**	**Memory**	**I/O**	**Init**	**Exec**	**Term**	**Freq**	**Clock**
param4	−	−	−	+	+	+	+	+
param5	+	+	+	−	−	−	+	+

Fig. 16. Speedup comparison of priority models.

D. Priority-Model Analysis

We now investigate to what extent the function priority model can affect the latency. Table IV shows the details of various configurations of two models under two traces and the results are illustrated in Fig. 16. We discover that in the Loglinear model, the speedup of param4 is $1.35\times$ (Azure) and $1.26\times$ (Huawei) higher than that of native, while the latency of param5 is 13% (Azure) and 4% (Huawei) lower than that of native. The reason is the setting of the resource weights. Param5 considers that the function with higher resource allocation is more critical. However, after allocating too many resources, the system cannot support more containers running at the same time, and the number of concurrency drops, further resulting in increased latency. In the linear model, param2 performs the best, followed by param1, with param3 being the least effective. Param2 obtains speedup ratios of 1.28 and 1.24 on two different traces. It takes into account all parameters, focusing on initialization time, termination time, and execution time. Param1 overlooks
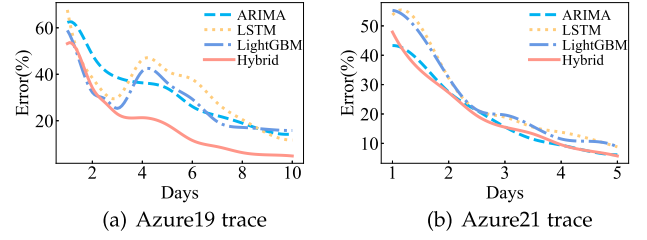

Fig. 17. Prediction model comparison.

I/O resources, concentrating instead on CPU and memory resources. Although its overall performance surpasses that of the native system, it is on average 9.5% lower than param2. Param3 disregards the function's resource information and emphasizes the importance of call frequency. Its performance is nearly equivalent to that of the native system. Comparing these two models, we find that extreme cases can lead to performance degradation although the Loglinear model can achieve the best result, i.e., param5. The average performance of the linear model is better than that of the loglinear model, but it is unable to access the optimal performance as param4 outperforms all the linear model parameters.

In summary, we should holistically set all parameters considering the sensitivity of the load to avoid huge performance discrepancies due to ignoring certain factors.

E. Predictor Performance Analysis

Here, we compare different time series prediction models using simulation under Azure 2019 and Azure 2021 traces. Fig. 17(a) shows the prediction results of 10 days. In the first three days, LightGBM and hybrid Prophet-LightGBM have the best prediction results. Since the fourth day is a special holiday, LightGBM and LSTM all undergo relatively large fluctuations, while hybrid Prophet-LightGBM, due to the characteristics of the model itself, adapts better to the data fluctuations. The final prediction error rates are 11.7% (LSTM), 14.3% (ARIMA), 15.8% (LightGBM), and 4.8% (Hybrid Prophet-LightGBM). The LSTM model does not adapt accurately to cyclical changes and special situations such as holidays. The LightGBM alone can not perform well without statistical features and prediction bounds while hybrid Prophet-LightGBM model is highly stable and accurate. Fig. 17(b) shows the results of 5 days of Azure 2021. Devoid of any special holidays, the prediction error shows a steady decline as data increases, while Incendio consistently maintains the lowest error rate (5.7%).

F. SLO Guarantees Analysis

We explore whether SLO can ensure fairness among requests. We first set a fixed SLO of 500 ms for four traces to investigate the latency distribution. As shown in Fig. 18, we find that Incendio can achieve at least 87% SLO guarantee, and can ensure 91% latency assurance on average. For the Huawei trace, 94% of requests can be scheduled within the SLO. We also conducted validation to ascertain the guarantee ratio of Incendio under various SLO standards across four traces, as depicted in
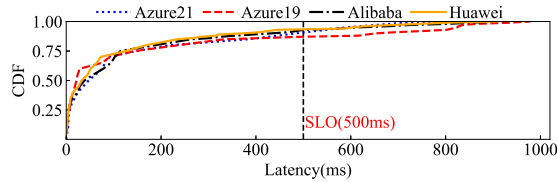
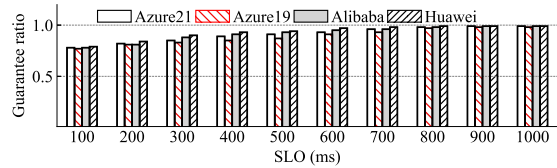Fig. 18.   SLO CDF guarantees under different traces.

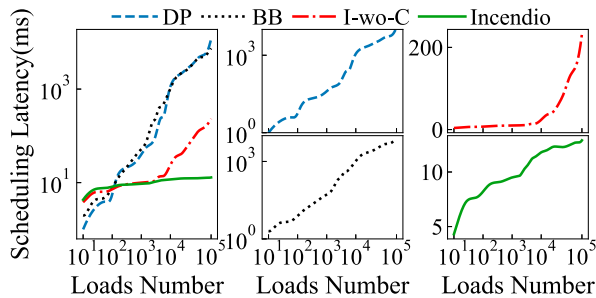Fig. 19.   SLO guarantees ratio under different limits.

Fig. 20.   Average scheduling latency. "DP", "BB" and "I-wo-C" denote dynamic programming, branch-and-bound and Incendio without clustering, respectively. The left part is the overall comparison, while the middle and right part show specific latency overheads.

Fig. 19. The results demonstrate that Incendio achieves an average ratio of approximately 80% at the lowest standard(100ms) and progressively increases to approximately 99% as the SLO threshold is elevated. These findings indicate that our scheduling approach can ensure fairness and prevent request starvation.

G. Scheduling Overhead Analysis

Finally, we conduct measurements to analyze the cost of scheduling policy. Since FPS is an NP-complete problem that can be solved by heuristics like Dynamic Programming (DP) and Branch and Bound (BB), we compare our CRL-based Incendio with the above approaches. We also conducted validation associated with traditional RL methods, specifically the native RL without clustering operations while maintaining all other settings consistent with CRL, i.e., Incedio without Clustering(I-wo-C). As shown in Fig. 20, both BB and DP experience latency that is 2-3 orders of magnitude higher than that of RL method as the number of loads increases. When the clustering operations are omitted, the traditional RL model also struggles to cope with heavy loads, exhibiting a tenfold increase in latency compared to CRL. On the other hand, CRL consistently maintains a latency of tens of milliseconds as the load increases.

IX. Related Work

In this section, we review the most relevant work with regard to the cold start in serverless systems, as well as its corresponding solutions in today's cloud platforms.

Warmup Runtime in Cache. Recent studies have shown the efficacy of reducing initialization times by caching containers or functions. Suo et al. [9] proposed HotC, which maintains a container pool in memory. The authors combined quadratic exponential smoothing and Markov chain to predict the live containers with historical data. Romero et al. [11] present Faa$T, an auto-scaling distributed cache. Upon reloading for the next invocation, Faa$T pre-warms the cache with objects likely to be accessed. The problem is that the cache hit is not totally equal to the warm start and is likely to waste memory. Our work differs from these efforts in that we focus on sensibly scheduling requests and managing runtime rather than solely relying on caching.

Priority-based Scheduling. Recent studies also valued the significance of functions, proposing similar perspectives on priority scheduling. Fuerst et al. [14] allocate priorities to containers to manage the container pool, evicting idle containers in order of priority. Mampage et al. [33] assume each request is associated with either a high or low priority level and allow for the eviction or re-scheduling of tasks. Tariq et al. [34] design a policy framework, including assigning priorities to functions and chains, which prevents lower priority functions from consuming resources needed by other functions. In comparison, our priority parameters are selected using Spearman's rank correlation coefficient and offer linear and log-linear models. This new approach allows for adjusting weight values based on system latency at runtimme, providing greater flexibility.

Overall latency in Serverless. Reducing the system latency remains a key issue in serverless. Gunasekara et al. [35] proactively spawn containers to avoid cold-start, thus minimizing the overall response latency. Yang et al. [36] propose INFless, featuring batch management and a novel Long-Short Term Histogram policy to guarantee overall latency. Singhvi et al. [37] design Atoll, which incorporates deadline-aware scheduling, minimizing sandbox setup to control request latency. While these methods are dedicated to reducing system latency, Incendio distinguishes itself through a scheduling and prewarming scheme.

Predict Serverless Workloads. Considerable research has focused on predicting the arrival of requests. Roy et al. [38] utilize a Fourier-transformation-based model to predict future invocation patterns in heterogeneous servers. Shahrad et al. [10] design a hybrid histogram strategy, combining historical histograms with ARIMA time series forecasting to prewarm containers or keep them alive. Mittal et al. [39] propose a lightweight regression-based incremental learning mechanism to train and predict the workload online. Compared to the aforementioned methods, our hybrid Prophet-LightGBM approach is equipped to adapt to sudden changes in request trends that typically occur during holidays, weekends, and special events.

Speedup   through   Checkpoint/Restore.   Much   recent research focused on recovery-based methods to mitigate the

cold start. Silva et al. [40] adopt checkpoint/restore-based methods, restoring snapshots from previously executed function processes. Ustiugov et al. [17] propose Record-And-Prefetch, recording functions' guest memory page and proactively prefetching it to address frequent page faults. The SnapStart technology [41] of AWS is based on Firecracker [42] and utilizes the snapshot feature of MicroVM. In contrast to speeding up checkpoint recovery, Incendio skips initialization by prewarming containers in advance.

Cost optimization of serverless service. With the increasing popularity of serverless, there's an increasing focus on optimizing service costs. Eismann et al. [43] optimize the billing model of Serverless and predict the optimal memory size for users through a multi-objective regression model. Elgmal et al. [44] optimize serverless workflow costs by determining whether to combine multiple functions into a larger function and the appropriate memory allocation. The above methods target service users, while our method focuses on reducing service latency from the cloud providers' perspective.

X. CONCLUSION

This paper presents Incendio, a new serverless framework that caters to the unique characteristics of functions and achieves optimal performance. Incendio efficiently schedules function requests via CRL and dynamically manages the runtime with the hybrid Prophet-LightGBM model, to ensure cost-optimality and latency-optimality. Experimental results on OpenWhisk show that Incendio can reduce up to 23% and 14.8% latency compared to two state-of-the-art methods. We believe that our approach offers an effective and practical mechanism for reducing the required resources for alleviating cold start in serverless computing.

REFERENCES

[1] "AWS Lambda." Amazon. Accessed: Apr. 15, 2024. [Online]. Available: https://aws.amazon.com/lambda/

[2] "Google cloud functions." Google. Accessed: Apr. 15, 2024. [Online]. Available: https://cloud.google.com/functions

[3] "OpenWhisk: Open source serverless cloud platform." Microsoft. Accessed: Apr. 15, 2024. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[4] "Azure functions." Apache. Accessed: Apr. 15, 2024. [Online]. Available: https://openwhisk.apache.org/

[5] "Openfaas is an independent open-source project." OpenFaas. Accessed: Apr. 15, 2024. [Online]. Available: https://www.openfaas.com/

[6] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surveys*, vol. 54, no. 10s, pp. 1–34, 2022.

[7] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proc. 6th USENIX Symp. Netw. Syst. Des. Implementation (NSDI)*, vol. 19, 2019, pp. 193–206.

[8] A. Mohan, H. S. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *Proc. 11th USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2019, pp. 3357034–3357060.

[9] K. Suo, J. Son, D. Cheng, W. Chen, and S. Baidya, "Tackling cold start of serverless applications by efficient and adaptive container runtime reusing," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 433–443.

[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. (USENIX ATC)*, USENIX, 2020, pp. 205–218.

[11] F. Romero et al., "Faa:$T: A transparent auto-scaling cache for serverless applications," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, New York, NY, USA: ACM, 2021.

[12] Y. Zhang et al., "Faster and cheaper serverless computing on harvested resources," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, 2021, pp. 724–739.

[13] M. Nazari, S. Goodarzy, E. Keller, E. Rozner, and S. Mishra, "Optimizing and extending serverless platforms: A survey," in *Proc. 8th Int. Conf. Softw. Defined Syst. (SDS)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 1–8.

[14] A. Fuerst and P. Sharma, "FaasCache: Keeping serverless computing alive with greedy-dual caching," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, New York, NY, USA: ACM, 2021, pp. 386–400.

[15] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "{ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation (OSDI 22)*, 2022, pp. 303–320.

[16] Z. Li et al., "Amoeba: QoS-awareness and reduced resource usage of microservices with serverless computing," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 399–408.

[17] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, New York, NY, USA: ACM, 2021, pp. 559–572.

[18] R. S. Venkatesh, T. Smejkal, D. S. Milojicic, and A. Gavrilovska, "Fast in-memory CRIU for Docker containers," in *Proc. Int. Symp. Memory Syst.*, 2019, pp. 53–65.

[19] K.-T. A. Wang, R. Ho, and P. Wu, "Replayable execution optimized for page sharing for a managed runtime environment," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–16.

[20] "AWS Lambda SLA." Amazon. Accessed: Apr. 15, 2024. [Online]. Available: https://aws.amazon.com/cn/lambda/sla/

[21] J. Kim and K. Lee, "FunctionBench: A suite of workloads for serverless cloud function service," in *Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 502–504.

[22] T. Yu et al., "Characterizing serverless platforms with serverlessbench," in *Proc. 11th ACM Symp. Cloud Comput. (SoCC)*, New York, NY, USA: ACM, 2020, pp. 30–44.

[23] Z. Wang et al., "Pigeon: An effective distributed, hierarchical datacenter job scheduler," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 246–258.

[24] P. Bogdan, "Mathematical modeling and control of multifractal workloads for data-center-on-a-chip optimization," in *Proc. 9th Int. Symp. Netw.-on-Chip*, 2015, pp. 1–8.

[25] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: Principled and practical scheduling for serverless functions," in *Proc. 13th Symp. Cloud Comput.*, 2022, pp. 289–305.

[26] "Cloud functions documentation." Google. [Online]. Available: https://cloud.google.com/functions/docs/configuring/memory

[27] "Optuna tool." Optuna. [Online]. Available: https://optuna.org/

[28] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, 2015, Art. no. 7540.

[29] D. Silver et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[30] J. Fan, Z. Wang, Y. Xie, and Z. Yang, "A theoretical analysis of deep Q-learning," in *Proc. 2nd Conf. Learn. Dyn. Control.*, PMLR, 2020, pp. 486–489.

[31] A. Wang et al., "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute," 2021, *arXiv:2105.11229*.

[32] A. Joosen et al., "How does it function? Characterizing long-term trends in production serverless workloads," in *Proc. ACM Symp. Cloud Comput.*, 2023, pp. 443–458.

[33] A. Mampage, S. Karunasekera, and R. Buyya, "Deadline-aware dynamic resource management in serverless computing environments," in *Proc. IEEE/ACM 21st Int. Symp. Cluster Cloud Internet Comput. (CCGrid)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 483–492.

[34] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling quality-of-service in serverless computing," in *Proc. 11th ACM Symp. Cloud Comput. (SoCC)*, New York, NY, USA: ACM, 2020, pp. 311–327.

[35] J. R. Gunasekaran, P. Thinakaran, N. Chidambaram, M. T. Kandemir, and C. R. Das, "Fifer: Tackling underutilization in the serverless era," 2020, *arXiv:2008.12819*.

[36] Y. Yang et al., "Infless: A native serverless system for low-latency, high-throughput inference," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2022, pp. 768–781.

[37] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A scalable low-latency serverless platform," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, New York, NY, USA: ACM, 2021, pp. 138–152.

[38] R. B. Roy, T. Patel, and D. Tiwari, "IceBreaker: Warming serverless functions better with heterogeneity," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2022, pp. 753–767.

[39] V. Mittal et al., "Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 168–181.

[40] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proc. 21st Int. Middleware Conf.,* New York, NY, USA: ACM, 2020, pp. 1–13.

[41] "AWS snapstart." Amazon. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html/

[42] A. Agache et al., "Firecracker: Lightweight virtualization for serverless applications," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation (NSDI)*, USENIX, 2020, pp. 419–434.

[43] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the optimal size of serverless functions," in *Proc. 22nd Int. Middleware Conf.*, 2021, pp. 248–259.

[44] T. Elgamal, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *Proc. IEEE/ACM Symp. Edge Comput. (SEC)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 300–312.

Yili Gong received the B.S. degree in computer science from Wuhan University, in 1998, and the Ph.D. degree in computer architecture from the Institute of Computing, Chinese Academy of Sciences, 2006. Currently, she is an Associate Professor with the School of Computer Science, Wuhan University. Her research interests include intelligent operations and maintenance in HPC environments, and distributed file systems.

Kun Suo received the B.S. degree in software engineering from Nanjing University, China, in 2012, and the Ph.D. degree from the University of Texas, Arlington, in 2019. Currently, he is an Assistant Professor with the Department of Computer Science, Kennesaw State University. His research interests include the areas of cloud computing, virtualization, operating systems, edge computing, Internet-of-things, and software-defined network.

Xinquan Cai received the B.S. degree in computer science and technology from ChongQing University, in 2021. He is currently working toward the Ph.D. degree in computer science with Wuhan University. His research interests include serverless computing and virtualization.

Qianlong Sang received the B.S. degree in cyber science and engineering from Wuhan University, in 2022. He is currently working toward the Ph.D. degree in computer science with Wuhan University. His research interests include edge computing and big data systems.

Xiaobo Zhou (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from Nanjing University, in 1994, 1997, and 2000, respectively. Currently, he is a Distinguished Professor with IOTSC and Department of Computer and Information Science, University of Macau, Macau S.A.R. His research interests include distributed systems and cloud computing. He serves as the Chair of IEEE Technical Community in Distributed Processing, from 2020 to 2023.

Chuang Hu (Member, IEEE) received the B.S. and M.S. degrees from Wuhan University, in 2013 and 2016, respectively, and the Ph.D. degree from Hong Kong Polytechnic University, in 2019. Currently, he is an Associate Researcher with the School of Computer Science, Wuhan University. His research interests include edge learning, federated learning/analytics, and distributed computing.

Dazhao Cheng (Senior Member, IEEE) received the B.S. and M.S. degrees in electrical engineering from Hefei University of Technology, in 2006, and the University of Science and Technology of China, in 2009, respectively, and the Ph.D. degree from the University of Colorado at Colorado Springs, in 2016. He was an Assistant Professor with the University of North Carolina at Charlotte, from 2016 to 2020. Currently, he is a Professor with the School of Computer Science, Wuhan University. His research interests include big data and cloud computing.