Online Container Caching for IoT Data Processing in Serverless Edge Computing

Guopeng Li, Haisheng Tan, *Senior Member, IEEE,* Chi Zhang, Xuan Zhang, Zhenhua Han, Guoliang Chen

***Abstract*—Serverless edge computing is an efficient way to execute event-driven, short-duration, and bursty IoT data processing tasks on resource-limited edge servers, using on-demand resource allocation and dynamic auto-scaling. In this paradigm, function requests are handled in virtualized environments, *e.g.*, containers. When a function request arrives online, if there is no container in memory to execute it, the serverless platform will initialize such a container with non-negligible latency, known as cold start. Otherwise, it results in a warm start with no latency in previous studies. However, based on our experiments, we find there is a remarkable third case called Late-Warm, *i.e.*, when a request arrives during the container initializing, its latency is less than a cold start but not zero. In this paper, we study online container caching in serverless edge computing to minimize the total latency with Late-Warm and other practical issues considered. We propose `OnCoLa`, a novel $O(T_c K)$-competitive algorithm supporting request relaying on multiple edge servers. Here, T_c and K are the maximum container cold start latency and the memory size, respectively. Extensive simulations on two real-world traces demonstrate that `OnCoLa` consistently outperforms the state-of-the-art container caching algorithms and reduces the latency by 23.33%. Experiments on Raspberry Pi and Jetson Nano show that `OnCoLa` reduces latency by up to 21.38% compared with the representative lightweight policy.**

***Index Terms*—Serverless computing, cache, data processing.**

I. Introduction

RECENTLY, the Internet-of-Things (IoT) has enabled new applications for many domains, including healthcare [2], public transport [3], and the energy industry [4]. These applications fundamentally rely on the processing of IoT data from IoT devices like sensors, cameras, and vehicles [5]. However, uploading IoT data to the remote cloud faces challenges like privacy leaks, network congestion, and high latency. As IoT data is often generated far from the cloud, edge computing is a natural alternative, which executes *IoT data processing tasks (IDPTs)* on edge servers close to the data sources [6]. IDPTs are event-driven, short-duration, and have bursty workloads [7]. When executing IDPTs on the resource (CPU, memory)-limited edge servers, it is challenging to handle the task bursts and prevent resource waste during idle periods between tasks. Adopting the serverless paradigm [8], which offers on-demand resource allocation and dynamic auto-scale policy, is a promising approach for executing IDPTs on resource-limited edge servers, which can be called *serverless edge computing* [9].

Serverless computing, also known as Function-as-a-Service (FaaS) has attracted attention from various communities, such as system [10]–[14], networking [15]–[17], and architecture [18]–[20]. In FaaS, developers implement tasks as *functions*, and execute functions within a virtualized environment, such as a container. Before executing, a container will go through an initialization, which involves launching the container, preparing the program language runtime, and installing necessary libraries. This initialization is known as a *cold start* with non-negligible latency. If the container is already initialized in memory before the function request, it results in a *warm start* with no latency. One way to mitigate cold starts is to cache initialized containers in memory so that they are warm for future function requests. However, caching containers in memory is costly, as about 50% of containers need more than 100 MB of memory [21]. Therefore, we need to investigate the container caching policy that decides which containers should be cached in memory to make full use of the limited memory and reduce the latency. Container caching is a fundamental problem in serverless computing, which is non-trivial due to variable memory demands, diverse cold start latencies, and skewed function popularity [21]. When taking into account the practical issues in edge computing, we reveal extra challenges as follows.

Late-Warm. In existing serverless edge computing studies [22]–[25], a function request either experiences cold start latency or incurs no extra latency (warm start). However, as shown in Fig. I, when a function request arrives during the corresponding container initialization, which strictly is neither a cold nor a warm start, we call this case *Late-Warm*, such as the requests for f_6 at T_2, T_3, and T_4. Late-Warm introduces a latency that is non-zero and shorter than a cold start. As shown in Fig. 2, this non-negligible latency makes the optimal policy (*i.e.*, Bélády [26]) for traditional online caching cannot be applied directly here. Moreover, our experiments on edge devices show that Late-Warm is more prevalent as the longer cold latency due to limited resources increases its occurrences.

Memory Sensitivity. Edge servers are resource-limited, *e.g.*, Raspberry Pi 4B (PI4B) and Jetson Nano (Nano) have CPU clock rates no higher than 1.5 GHz and main memory no more than 8 GB. Therefore, as shown in our experiments in Example 1, we observed significant variations in cold start

This work was supported in part by the National Key R&D Program of China under Grant 2021ZD0110400, NSFC under Grant 62132009, and the Fundamental Research Funds for the Central Universities at China. A preliminary version with part of the results has been accepted by the 2024 IEEE 40th International Conference on Data Engineering (ICDE), Utrecht, Netherlands [1]. Guopeng Li, Haisheng Tan, Xuan Zhang, and Guoliang Chen are with University of Science and Technology of China, Hefei 230026, China (e-mail: guopengli@mail.ustc.edu.cn; hstan@ustc.edu.cn; xuanzhang@mail.ustc.edu.cn; glchen@ustc.edu.cn), Chi Zhang is with Hefei University of Technology, Hefei 230009, China (e-mail: zhangchi@hfut.edu.cn), Zhenhua Han is with Microsoft Research Asia, Shanghai 200232, China (e-mail: Zhenhua.Han@microsoft.com). Corresponding author: Haisheng Tan.

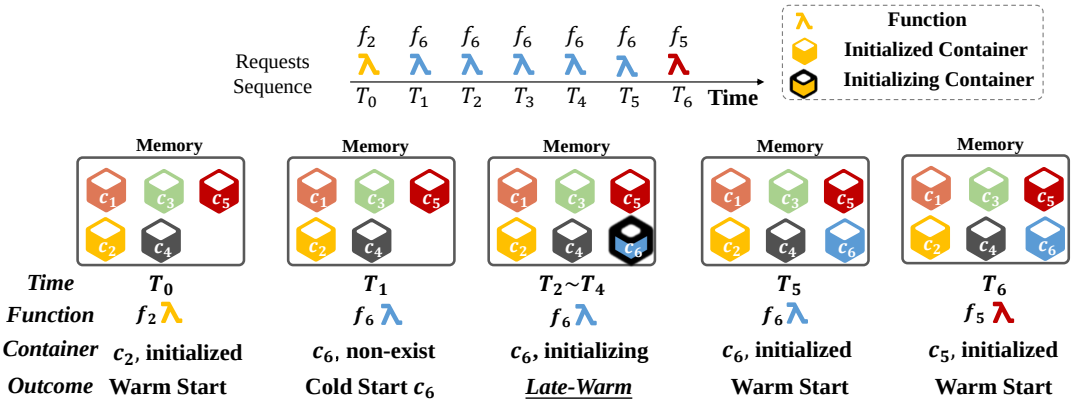


Fig. 1. Late-Warm in serverless edge computing. c_i represents the container used to execute f_i, for $1 \le i \le 6$. Before T_0, containers $c_1 \sim c_5$ have been initialized in memory. c_6 starts initializing (*i.e.*, cold start) at T_1 until T_5, and therefore the outcomes of requests arriving at T_2,T_3, and T_4 are all *Late-Warm*.

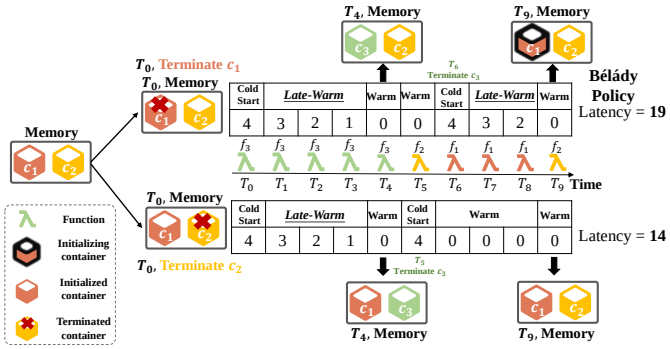
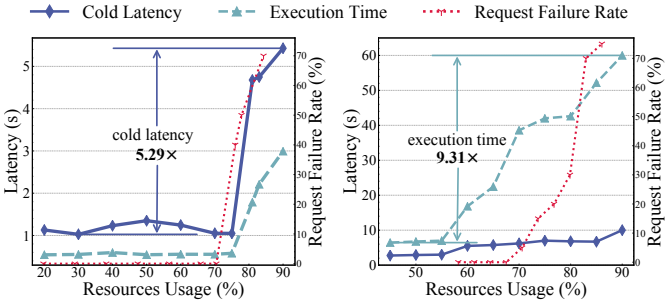


Fig. 2. Bélády is latency-suboptimal in online caching with Late-Warm. In this example, the size of memory is 2, for any container, the memory footprint is 1, and the cold latency is 4, that is, the initialization starts at T_i and finishes at T_{i+4}. At T_0, since c_3 requires memory for initialization, one must choose to terminate either c_1 or c_2. Bélády terminates c_1. The result shows that terminating c_1 leads to a latency of 19, while terminating c_2 results in a latency of 14. Therefore, Bélády is latency-suboptimal.

latency and function execution time with different memory usage percentages. Additionally, a container's memory footprint changes significantly depending on whether it is actively executing, as described in Sec. III and Table II. Unlike traditional online caching, which mainly focuses on files with fixed retrieval latency and memory footprint [27]–[29], this paper focuses on container caching, where cold latency, execution time, and container memory footprint are much more dynamic. These dynamics in latency and container memory footprint introduce new challenges for container caching. Moreover, with a high memory usage percentage, function requests might even result in failures.[1] As illustrated in Fig. 3, the latency and request failure rate might sharply increase at some specific memory usage in both PI4B and Nano, further inspiring us to avoid such a sudden increase adaptively when designing the container caching policy.

Example 1 (Memory Sensitivity)**.** Fig. 3 shows our experimental results on PI4B with 1GB memory and Nano with 4GB memory. We invoke a matrix multiplication function on PI4B and an image classification function on Nano. When memory usage percentage changes, the cold start latency and the function execution time can vary by up to 5.29× and 9.31×, respectively. Moreover, the latency and request failure have a sharp increase when memory usage reaches nearly 80% and 70% for PI4B and Nano, respectively. We also conducted experiments on the influence of CPU usage on latency and failure rate, and found it much less sensitive compared with the factor of memory usage. Due to limited space, we omit the result here.



(a) Matrix multiplication on PI4B (b) Image classification on Nano

Fig. 3. The variation of cold start latency, function execution time, and request failure rate with the change of Memory usage percentage.

Request Relaying. In serverless edge computing, function requests arrive at each edge server in a distributed manner. When an infrequent function request arrives on one edge as a cold start, a more cost-effective approach might be to not initialize its container locally, but to send it to another server that has the container with additional relay latency, which is called *request relaying*. This becomes especially challenging with the additional latency from Late-Warm, the variable cold latency and execution time caused by Memory Sensitivity and the relay latency between servers, elevating the difficulty of decision-making in the online container caching algorithm.

To address the above practical challenges in serverless edge computing for executing IoT data processing tasks, we study the online container caching problem with Late-Warm on multiple edge servers. We propose a novel online algorithm named `OnCoLa` to minimize the total latency of function requests. In `OnCoLa`, we assign a priority to each container to indicate its cost-effectiveness for reducing the total latency. We implement

[1]If there is insufficient memory to initialize a container for a new function on any server and terminating executing containers is not permitted, or if the function's execution memory requirements exceed available memory, this will result in a request failure.

and evaluate it with small-scale testbed experiments using common IoT data processing tasks and large-scale simulations based on real-world traces. Our technical contributions are summarized as follows:

- Under a novel model taking Late-Warm and other practical issues into account in executing IoT data processing tasks on edge servers, we investigate the online container caching problem on multiple edge servers to minimize the total latency. We analyze its hardness and prove the lower bound of the competitive ratio as $\Omega(T_c K)$, where T_c is the maximum cold start latency, and K is the memory size. To the best of our knowledge, we are the first to explicitly consider Late-Warm and the relay latency for the online container caching problem (in Sec. III).
- We propose an Online Container Caching policy with Late-Warm, named `OnCoLa`, taking Late-Warm, memory sensitivity, and request relaying into account. We further theoretically prove its competitive ratio as $O(T_c K)$ (in Sec. IV).
- Through extensive large-scale simulations with AliFC trace and Azure trace, we demonstrate that `OnCoLa` outperforms the SOTA solution GD [30] and reduces the latency by up to 23.33%. We implement `OnCoLa` on PI4B and Nano with OpenFaaS and faasd, and evaluate it under workloads consisting of common IoT data processing tasks. The results demonstrate that `OnCoLa` significantly reduces latency by 21.38% and reduces request failure rate by up to 2.3× compared with the commonly used fixed-duration container caching policy (in Sec. V and Sec. VI).

II. Background

A. IoT Data Processing in Serverless Edge Computing

Serverless computing offers a Function-as-a-Service (FaaS) abstraction, enabling tasks to be deployed as serverless functions that are invoked by events such as request arrivals or new data production. These serverless function requests are managed in virtualized environments like containers [31]. The event-driven execution paradigm and dynamically auto-scaling resource management policy of serverless computing motivate us to apply serverless to edge computing for executing event-driven and bursty IoT data processing tasks [9], where data is processed on edge servers near the source, reducing latency and bandwidth usage. Fig. 4 illustrates the components for executing IoT data processing tasks in serverless edge computing, including IoT devices like sensors, cameras, and smartphones, and edge servers PI4B and Nano. In this setup, when an IoT device generates a data processing event, it triggers a request for a specific function (such as Object Detection, text-to-speech, etc.). Upon receiving this request, PI4B or Nano executes the function by initializing a container.

B. Cold Start

Cold start affects serverless computing both in the cloud and at the edge, introducing additional latency. As an example, Fig. 5 illustrates the latency composition for a request to an image classification function executed on a Jetson Nano. The cold start process typically involves stages such as checking for an available initialized container, starting the container instance, preparing the language runtime environment, and importing necessary libraries. In the illustrated case, the extra latency from the cold start is 2.75 seconds, compared to a function execution time of 6.45 seconds. This additional cold start latency accounts for 29.89% of the total end-to-end latency. Such latency is critical in applications requiring low latency, like real-time data processing and interactive services, potentially degrading performance and user experience. Consequently, optimizing serverless computing necessitates cold start mitigation. There are two primary strategies for this: reducing the frequency of cold starts and decreasing the duration of individual cold starts, for instance, by accelerating container initialization. In this paper, we focus on the first. These two methods are orthogonal and complementary in achieving the overall goal of cold start mitigation.

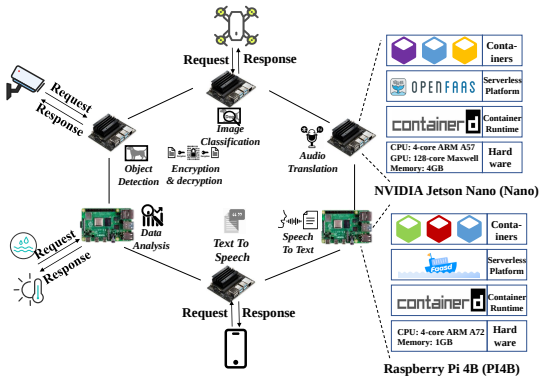


Fig. 4. IoT Data Processing in serverless edge computing.

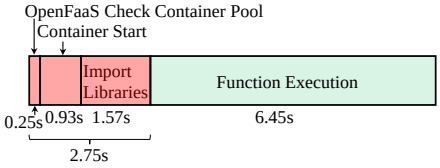


Fig. 5. The composition of latency.

III. System Model and Problem Formulation

We provide the system model and problem formulation in this section. Commonly used symbols are listed in Table I.

A. Model

System. Motivated by serverless edge computing, this study focuses on a system comprising multiple edge servers. Specifically, the system consists of N edge servers, $\mathcal{S} = \{s_1, s_2, \ldots, s_N\}$, where the memory size of each server is $K_i, i = 1, 2, \ldots, N$. We set $K = \max_i K_i$. The request of function $f_i \in \mathcal{F}$ ($\mathcal{F} = \{f_1, f_2, \ldots\}$) is assumed to execute in its own container c_{f_i}. Whenever a function f_i is requested, its corresponding container c_{f_i} needs to be initialized to execute the function. We use t^{re} to denote the relay latency, defined as the time required to relay a request from the requested edge server to another.

Container. A container in memory can be in one of three states at any time: *initializing*, *initialized*, or *executing*. An *initializing* container is a container that has not finished its initialization and cannot execute any functions. An *initialized* container is a container that has been initialized, but has no

function requests at the moment. It can also be called an idle container. An *executing* container is a container that is executing a function.[2] A container that is initialized or executes a function is called a *warm container*, which represents an already initialized environment for the requests of the same function. Generally, we use z_{f_i} to represent the memory footprint of container c_{f_i}. Specifically, $z^e_{f_i}$ denotes the memory footprint when the container is in the executing state, and $z^p_{f_i}$ represents the footprint when the container is in the initializing or initialized state. For convenience, we use c_f to represent c_{f_i}, z_f to denote z_{f_i}, z^e_f to denote $z^e_{f_i}$, and z^p_f to denote $z^p_{f_i}$. The sum of the container sizes on each edge server must not exceed its memory capacity.

TABLE I
LIST OF SYMBOLS

Notation	Description
c_f	The corresponding container to execute function f.
t^e_f	The function execution time of f.
t^c_f	The latency for initializing container c_f.
t^{re}	The latency for relaying one request from one edge server to another edge server.
p_{c_f}	The priority of container c_f.

Request. Let $\mathcal{R} = (r_1, r_2, \dots)$ be the sequence of function requests. We represent a request as a pair $(s, f) \in \mathcal{S} \times \mathcal{F}$, meaning the request of function f on edge server s. All function requests arrive in an online manner, that is, we can not get future information and we make no assumptions on the arrival patterns. We divide time into slots of unit size. Multiple different kinds of function requests might come within one time slot, however, each function $f \in \mathcal{F}$ can be requested at most once in each slot. We use t^e_f to indicate the execution time of f, and t^c_f to indicate the latency for initializing the container c_f (*i.e.*, cold latency), and t^e_f and t^c_f vary with memory usage. We set $T_c = \max_i t^c_{f_i}$. As shown in Fig. III-A, there are four different outcomes for processing request $r := (s, f)$ based on the state of c_f, and resulting in different latency:

- *Cold Start* (*e.g.*, r_2): If c_f is not in the memory of s, one option is to initialize a new container on s, known as Cold Start. If there is insufficient memory available, containers will be terminated[3]. The latency for processing request r consists of the execution time of function f and the initialization latency, represented as $t^e_f + t^c_f$.
- *Late-Warm* (*e.g.*, r_4): If the state of c_f in the memory of edge server s is *initializing*, we call it Late-Warm. The latency for processing request r includes the execution time of function f and the waiting time for c_f to finish initializing, denoted as $t^e_f + t^q_f$, where $0 < t^q_f < t^c_f$.
- *Warm Start* (*e.g.*, r_3): When there is an already initialized container c_f for the request of f, it is known as a Warm Start. The latency for processing this request is the execution time of f, t^e_f.
- *Relay* (*e.g.*, r_1): If c_f does not exist in the memory of edge server s, but there is an initialized c_f on another edge server, s', one option is to relay request r to s' for processing, referred to as Relay. The latency for processing r would then be the relay latency and the time to execute function f, denoted as $t^{re} + t^e_f$.

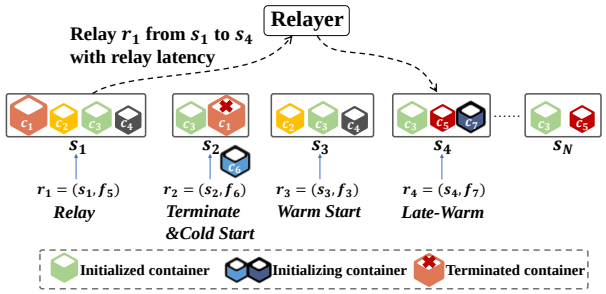


Fig. 6. Container caching on edge servers with 4 different cases: Relay, Terminate & Cold Start, Warm Start and Late-Warm.

B. Problem Formulation

The objective of this problem is to minimize the total latency for processing all function requests. Let $t_{r:=(s,f)}$ denote the latency incurred by processing the request $r := (s, f)$.

Problem P:

$$\begin{aligned} &\text{minimize} \quad \sum_{r \in \mathcal{R}} t_{r:=(s,f)} \\ &s.t. \quad \sum_{c_f \text{ in server } s_i} z_f \le K_i \qquad \forall i \in \{1, 2, \dots, N\} \end{aligned} \tag{1}$$

For the hardness of Problem P, its simplified version, where each container is of uniform size, has been proved NP-Complete [32]. Further, we have the following theorem.

Theorem 1. *All online algorithms for Problem P have a competitive ratio lower bound as* $\Omega(T_c K)$.

Proof. We use *pure* and *bursty* requests. A pure request for f_i on server s has $T_c + 1$ slots, with f_i on s in the first slot and no requests in the rest. A bursty has $2T_c$ slots, with f_i on s in the first T_c slots and no requests in the rest. The latency is t^e_f for warm pure requests, $t^e_f + t^c_f$ for cold pure requests, $T_c t^e_f$ for warm bursty requests, and $T_c t^e_f + t^c_f(t^c_f + 1)/2$ for cold bursty requests. Let r^p_i and r^b_i be pure and bursty requests for f_i. Assume $K+1$ different functions are requested. Let $\mathcal{A}$ be an online algorithm for problem P. We assume that the containers of functions $f_1, \dots, f_K$, *i.e.*, $c_{f_1}, \dots, c_{f_K}$ have been initialized initially. The constructor first pure requests r^p_{K+1}, which terminates one container from $c_{f_1}, \dots, c_{f_K}$. Then it repeats bursty requests for K times. The j-th bursty request is $r^b_{i_j}$, where c_{f_j} is the terminated container before the request. So, for $\mathcal{A}$, each bursty request has latency $T_c t^e_f + t^c_f(t^c_f + 1)/2$, and the total latency of $\mathcal{A}$ is $t^c_f + t^e_f + K(T_c t^e_f + t^c_f(t^c_f + 1)/2)$, the total cold latency is $t^c_f + K t^c_f(t^c_f + 1)/2$. However, for the optimal algorithm, the total latency is $t^c_f + t^e_f + K T_c t^e_f$, and the total cold latency is t^c_f. Therefore, the competitive ratio is bounded by $\Omega(T_c K)$. □

[2]In serverless computing, multiple concurrent requests of the same function can be handled in a single container or by initializing multiple containers, depending on the auto-scaling policy of the serverless platform.

[3]If all the containers are executing and not allowed to terminate, resulting in a request failure.

IV. Online Algorithm

In this section, we introduce OnCoLa, an online container caching algorithm that supports relaying on multiple edge servers with Late-Warm. OnCoLa tackles the challenges of Late-Warm, Memory Sensitivity, and Request Relaying through three key components. First, it employs a priority to quantitatively assess the cost-effectiveness of caching each container while considering Late-Warm, guiding termination decisions when memory is insufficient. Second, an Admission Policy determines whether to initiate a new container at the requested edge server locally or relay the request to another server that already has an initialized container. Third, OnCoLa incorporates a Memory Adjustment method to dynamically manage memory thresholds, preventing sudden increases in latency and request failures. The subsequent subsections detail the framework (Sec. IV-A), the priority calculation (Sec. IV-B), the admission policy (Sec. IV-C), and the memory adjustment method (Sec. IV-D).

A. Online Container Caching on Multiple edge servers

In the online container caching problem on multiple edge servers, the key challenges involve relaying or locally processing requests and selecting containers to terminate under memory insufficiency. In OnCoLa, we assign a priority, p_{c_f}, to each container c_f, to represent its cost-effectiveness in reducing total latency.

Alg. 1 shows the details of OnCoLa, for an online arriving function request $r := (s, f)$, based on the state of c_f, and each container's priority, executes r result in Warm, Late-Warm, Cold Start or Relay. Initially, the memory for caching containers is empty (Line 2). At any time T, the container states are updated, and it is checked if all buffered requests for f on s can be executed (Lines 6 to 11). When a new request $r := (s, f)$ for function f arrives at s, the state of container c_f on s is checked. If the state is INITED, *i.e.*, the state of c_f is initialized or executing, it is a Warm start (Line 14). If c_f is still initializing, it is a Late-Warm (Line 16). When c_f does not exist on s, according to the result obtained by the Admission Policy, determine whether to initialize c_f on s or relay r to another edge server s'. The details of the Admission Policy are in Alg. 3. If the result returned by Admission(s, f) is s, and there is insufficient memory to start a new container, the lowest priority container(s) are terminated to release memory (Line 21 to 23). The priority of each container is decreased by p_{min}, *i.e.*, decrease the priorities of the containers cached in memory but not currently requested (Line 24). Then c_f is initialized on s, which is a Cold Start (Lines 26 to 28). If Admission(s, f) returns s', then relay r to s'(Lines 29 to 30).

B. Priority

In the online container caching problem on edge servers, since containers have varied cold start latency, execution time, and memory footprints, it is crucial to evaluate how caching different containers impacts the total latency quantitatively. In OnCoLa, we address this by assigning a priority p_{c_f} to each container c_f to indicate its cost-effectiveness, represented as the ratio of latency reductions to memory footprint. The intuition behind the priority is OnCoLa prefers to cache those containers that offer greater latency reduction per unit of memory used in memory. The priority of the container not in memory is 0. When faced with memory insufficient, OnCoLa terminates the container with the lowest priority, allowing for greater latency reduction with less memory use. The method to calculate p_{c_f} is detailed in Eqn. 2, incorporates estimates of latency reduction and memory footprint as the numerator and denominator, respectively.

As shown in Eqn. 2, the numerator estimates the latency reduction from caching c_f by using γ, $t_f^{c\,2}$ and c_f.AvgLate. $t_f^{c\,2}$, a simplified term representing the potential maximum latency

Algorithm 1: OnCoLa

```
1  Input Request r := (s, f), Priority p_{c_f}, z_f = z_f^e ;
2  C ← ∅, C represents the containers cached in the
     memory of s;
3  Initializing containers C_Init ← ∅, (s, c_f, t) ∈ C_Init
     means container c_f will be fully initialized on s at
     time t;
4  Timer T ← 0;
5  while True do
6      for (s, c_f, t) ∈ C_Init do
7          if t <= T then
8              if c_f.state = INITING then
9                  c_f.state ← INITED;
10                 C ← C ∪ {c_f};
11             Serve all the buffered requests for f on s;
12     while new request r := (s, f) for function f on s
         arrive at T do
13         if c_f.state = INITED then // Warm
14             Execute f in c_f on s with latency t_f^e;
15         if c_f.state = INITING then // Late-Warm
16             Execute f in c_f on s at time t with latency
                 t − T + t_f^e;
17         if c_f.state = OUT then
18             if Admission(s, f)==s then
19                 while remain size of s < z_f do
20                     if p_{c_f'} is the lowest priority then
21                         p_min = p_{c_f'};
22                         Terminate c_f' on s, C \ {c_f'};
23                         c_{f'_state} ← OUT;
24                 For container c_f ∈ C, p_{c_f} = p_{c_f} − p_min;
25                 p_min ← 0;
26                 c_f.state ← INITING// ColdStart;
27                 C_Init ← C_Init ∪ {(s, c_f, T + t_f^c)};
28                 Initializing c_f on s with t_f^e + t_f^c ;
29             else // Admission(s, f) returns s'
30                 Relay r := (s, f) to s' with latency
                     t_f^e + t^re;
31         UpdatePri(s, f);
32     T ← T + 1;
```

savings derived from the worst-case scenario considering Late-Warm effects, and c_f.AvgLate is the dynamically calculated average latency actually observed during previous cold starts for c_f, which accounts for variable latencies and non-worst-case request patterns. This combination aims to provide a more robust estimate than using either term alone. The denominator, z_f, estimates the container's memory footprint using both static initialized (z_f^p) and executing size (z_f^e), as a weighted average ($z_f = R_{\text{run}} \cdot z_f^e + \max(1 - R_{\text{run}}, 0) \cdot z_f^p$) reflecting the proportion of time (R_{run}) spent executing. The rationale for calculating these estimations is detailed as follows.

$$p_{c_f} = \frac{(1-\gamma) \cdot t_f^{c\,2} + \gamma \cdot c_f.\text{AvgLate}}{z_f}. \tag{2}$$

$t_f^{c\,2}$: Before the container c_f is fully initialized, requests for function f result in Late-Warm instead of Warm Start, with a maximum waiting time of t_f^c. So, for a Cold Start, in the worst case, if t_f^c requests for f within the next t_f^c time, the total latency is $(t_f^c+1)t_f^c/2$. This means that if c_f is cached in memory, the maximum latency reduction could be $(t_f^c+1)t_f^c/2$. For simplicity, we use $t_f^{c\,2}$ to estimate the latency reduction for caching c_f in memory.

c_f.AvgLate: As Fig. 3 shows, the cold latency t_f^c varies, and due to the skewed popularity of function requests, not all requests experience the worst case. Hence, using only $t_f^{c\,2}$ to estimate the latency reduction for caching c_f in memory is not appropriate. We calculate c_f.AvgLate in Alg. 2 (Lines 3 to 8), to represent the average latency caused by a cold start of c_f during online function request processing. This value is based on the actual latency observed during online execution rather than a fixed value. While $t_f^{c\,2}$ reflects the worst case, to incorporate both worst case and online execution, we compute c_f.cost $= (1-\gamma) \cdot t_f^{c\,2} + \gamma \cdot c_f$.AvgLate as the estimation of latency reduction of caching c_f in memory, using γ to balance between the two methods.

z_f: We use z_f to estimate the memory footprint of container c_f. When a container is initialized but not executing, its memory footprint z_f^p is the minimum required to store its virtual environment configuration and metadata. When a container is executing, its memory footprint z_f^e depends on the function code running inside. Our experiments on Nano show that the difference between z_f^e and z_f^p can be up to 100×, as Table II shows. Therefore, it is not appropriate to use z_f^e or z_f^p alone as the memory footprint of c_f, but instead, we use z_f. Specifically, $z_f = R_{\text{run}} \cdot z_f^e + \max(1 - R_{\text{run}}, 0) \cdot z_f^p$. Here, R_{run} represents the proportion of time that c_f has been in the executing state since it was fully initialized.[4]

C. Container Admission Policy

For request $r := (s, f)$, when there is no initialized container c_f on s, it is a problem whether to initialize c_f on s or relay r to another edge server. For this, as part of OnCoLa, we design an admission policy to decide whether to admit initializing c_f on s. Alg. 3 shows the details of the admission policy. In

[4] R_{run} can exceed 1 when a container in the executing state processes multiple function requests for the same function by forking new processes, such as "fork fprocess" in OpenFaaS [33].

Algorithm 2: UpdatePri

```
1  Input Edge Server s, function f
2  if c_f.state = OUT then
3      c_f.cumLate ← c_f.cumLate + t_c^f;
4      c_f.numLate ← c_f.numLate + 1;
5  if c_f.state = INITING then
6      c_f.cumLate ←
         Total Late-Warm latency of all buffered requests;
7      c_f.numLate ← c_f.numLate + 1;
8  c_f.AvgLate = c_f.cumLate / c_f.numLate;
9  c_f.cost ← (1 − γ) · t_f^{c 2} + γ · c_f.AvgLate;
10 R_run = Total time of c_f executing f / Duration since c_f initialized;
11 z_f = R_run · z_f^e + max(1 − R_run, 0) · z_f^p;
12 p_{c_f} = c_f.cost / z_f;
```

Alg. 3, returning s means admitting the initialization of c_f on s, and returning s' means relaying request r to another edge server. First, if there is no already initialized c_f on any servers, obviously, initializing c_f on s is the only choice, Alg. 3 returns s (Line 8). If there is one already initialized c_f on other servers, the decision whether to initialize c_f on s (Alg. 3 returns s) or relay r to another edge server (Alg. 3 returns s') is based on the comparison between p_c^f and $\frac{\max(0, t_c^f - t^{re})}{z_f^e}$. Here, p_c^f is the priority of container c_f on edge server s, t_c^f is the cold latency for initializing c_f, t^{re} is the relay latency, and z_f^e is the memory footprint when c_f is in the executing state. That is, we use $\frac{\max(0, t_c^f - t^{re})}{z_f^e}$ to indicate the "priority" of relaying r to another edge server. If $p_c^f \le \frac{\max(0, t_c^f - t^{re})}{z_f^e}$, then return s', otherwise, it returns s, admitting the initialization of c_f on s. It is worth noting that if the container c_f has never been initialized on s, its priority is 0.

Algorithm 3: Admission

```
1  Input Edge Server s, function f
   Output: Edge Server
2  if there is a server s' has c_f then
3      if p_{c_f} ≤ max(0, t_c^f − t^{re}) / z_f^e then
4          return s'
5      else
6          return s
7  else
8      return s
```

D. Memory Adjustment

As a component of OnCoLa, we propose a conservative *memory adjustment method* to prevent sudden increases in t_f^e, t_f^c, and request failures. This method comprises two components: profiling memory thresholds for different servers, and

enabling *memory growth* on servers experiencing dense request arrivals to prevent overly conservative memory thresholds. Specifically, we profile the memory usage percentage M_{th} of different servers during sudden increases, defined as more than a 20% growth in t_f^e, t_f^c, or request failures. We set the memory size K_i^a for caching containers on server s_i as $K_i \cdot M_{\text{th}}$. Here, M_{th} is referred to as the initial *memory threshold*. To adjust the memory size online, we maintain a ghost list [34]. When container c_f on server s is terminated, we add f to the ghost list (without keeping c_f in memory). If $r = (s_i, f)$ re-arrives within the cold start latency t_f^c and before this, there have been no sudden increases in t_f^e, t_f^c, and request failures, we increase K_i^a by z_f^e if $K_i^a + z_f^e \leq K_i$, we call it *memory growth*.

E. Theoretical Analysis

Lemma 1. *`OnCoLa` is $O(K)$-competitive for container caching without Late-Warm.*

Proof. We use the potential function method [35] to prove this Lemma. `OPT` is the optimal algorithm. We define the potential function as follows:

$$\Phi = (K-1) \cdot \sum_{c_f \in \text{Mem}} p_{c_f} + K \cdot \sum_{c_f \in \text{OptMem}} p_{c_f}^{init} - p_{c_f}$$

Here Mem and OptMem indicate the memories of `OnCoLa` and `OPT`, respectively. For containers not in memory, $p_{c_f} = 0$, and $p_{c_f}^{init}$ is the priority of c_f when it starts cold start. Initially, Φ is zero, and finally, $\Phi \geq 0$, satisfying the requirements of a potential function. For each request, we have:

- If `OnCoLa` initializes a container with $p_{c_f}^{init}$, Φ decreases by at least $p_{c_f}^{init}$.
- If `OPT` initializes a container with $p_{c_f}^{init}$, Φ increases by at most $K \cdot p_{c_f}^{init}$.
- Otherwise, Φ does not increase.

These facts imply that the cost incurred by `OnCoLa` is bounded by K times the cost incurred by `OPT`.

Next, we analyze in detail the impact of different cases on Φ after receiving one request.

- `OPT` terminates a container c_f: since $p_{c_f}^{init} - p_{c_f} \geq 0$, Φ does not increase.
- `OPT` initializes a container c_f: `OPT` pays $p_{c_f}^{init}$. Since $p_{c_f}^{init} \geq 0$, Φ increases by at most $K \cdot p_{c_f}^{init}$.
- `OnCoLa` reduces the priority for all containers in Mem: Since the decrease of a given priority p_{c_f} is p_{min}, the decrease in Φ is $p_{min} \cdot ((K-1) \cdot n_{\text{Mem}} - K \cdot n_{\text{Mem} \cap \text{OptMem}})$. We use n to denote the number of containers in memory. When this case occurs, it indicates that there is not enough memory available in Mem to initialize c_f. And we can assume that c_f has already been initialized in OptMem. Therefore, Mem$\cap$OptMem $\neq$ Mem and $n_{\text{Mem}} - n_{\text{Mem} \cap \text{OptMem}} \geq 1$. Thus, the decrease in Φ is at least $p_{min} \cdot (K \cdot (n_{\text{Mem}} - n_{\text{Mem} \cap \text{OptMem}}) - n_{\text{Mem}})$. Since $n_{\text{Mem}} - n_{\text{Mem} \cap \text{OptMem}} \geq 1$, and the memory size $K \geq n_{\text{Mem}}$, Φ decreases by at least 0.
- `OnCoLa` terminates a container c_f: Since the terminated container has the least priority p_{min}, Φ does not increase.
- `OnCoLa` realying the request from s to s': Φ is unchanged.
- `OnCoLa` initializes the request container c_f and sets $p_{c_f}^{init}$: The cost of this step is $p_{c_f}^{init}$. This container was not in Mem before, and we assume it has been initialized in OptMem, Φ decreases by $-(K-1)p_{c_f}^{init} + Kp_{c_f}^{init} = p_{c_f}^{init}$.

Thus, the cost incurred by `OnCoLa` is bounded by K times the cost incurred by `OPT`, `OnCoLa` is $O(K)$ - competitive for container caching without Late-Warm. □

Theorem 2. *`OnCoLa` is $O(T_c K)$-competitive for container caching with Late-Warm.*

Proof. We define some notations for this proof. Let $\text{ALG}(t_f^c)$ and $\text{OPT}(t_f^c)$ be the total latency of `OnCoLa` and the offline optimal in the online container caching model with Late-Warm. Let $\text{MALG}(t_f^c)$ and $\text{MOPT}(t_f^c)$ be the total cost of the online algorithm MALG and the offline optimal of online container caching on multiple edge servers, where MALG is c-competitive and t_f^c is the cost to start c_f. We have $\text{MALG}(t_f^c) \leq c \cdot \text{MOPT}(t_f^c)$. In the proof, we use t_c to present t_f^c and t_e to present t_f^e.

1. $\text{ALG}(t_c) \leq \text{MALG}(t_c^2)$.

We define the request sequence of a function as all requests to f from the cold start of c_f to the next cold start or a relaying for f. Each request sequence of f has one cold start of c_f and zero or more Late-Warm of f. Each initialization of f causes at most $t_c - 1$ Late-Warm, so the initialization latency of each request sequence of f of $\text{ALG}(t_c)$ is at most $\frac{t_c \cdot (t_c+1)}{2}$. The initialization cost of each request sequence of f of $\text{MALG}(t_c^2)$ is t_c^2. For each relaying of f, the latency of f in `OnCoLa` is $t^{re} + t_e$, and the cost of f in MALG is $t^{re} + t_e$. Thus, $\text{ALG}(t_c) \leq \text{MALG}(t_c^2)$.

2. $\text{MOPT}(t_c^2) \leq T_c \cdot \text{MOPT}(t_c)$.

In the model of online container caching on multiple edge servers, let $\mathcal{S}_1$ and $\mathcal{S}_2$ request the same functions, with initialize costs $(w_1, w_2, \ldots, w_n)$ and $(\alpha w_1, \alpha w_2, \ldots, \alpha w_n)$ in $\mathcal{S}_1$ and $\mathcal{S}_2$. Then $\text{MOPT}(\mathcal{S}_1) = \alpha \cdot \text{MOPT}(\mathcal{S}_2)$. If $(w_1, w_2, \ldots, w_n)$ and $(w_1', w_2', \ldots, w_n')$ are the cold start costs in $\mathcal{S}_1$ and $\mathcal{S}_2$, and $w_i \leq w_i'$ for all i, then $\text{MOPT}(\mathcal{S}_1) \leq \text{MOPT}(\mathcal{S}_2)$. So, $\text{MOPT}(t_c^2) \leq T_c \cdot \text{MOPT}(t_c), T_c = \max t_c$.

3. $\text{MOPT}(t_c) \leq \text{OPT}(t_c)$. We set $\mathcal{B}$ as an algorithm that meets the following conditions: Firstly, $\mathcal{B}$ and OPT have the same main structures such as container terminal decision. Secondly, $\mathcal{B}$ and OPT differ only in how they compute total latency. For the j-th request of function f_i, we assume the time slot when this request arrives is $T_{i,j}$. Besides, we define the increase of the total latency caused by the j-th request of function f_i in OPT be $\text{OPT}_{i,j}$ and define the increase of the total latency caused by this request in $\mathcal{B}$ be $\mathcal{B}_{i,j}$. If the outcome of the j-th request of f_i is warm start, $\mathcal{B}_{i,j} = \text{OPT}_{i,j} = t_e$. If the outcome of the j-th request of f_i is cold start, $\mathcal{B}_{i,j} = \text{OPT}_{i,j} = t_c + t_e$. If the outcome of the j-th request of f_i is relay, $\mathcal{B}_{i,j} = \text{OPT}_{i,j} = t^{re} + t_e$. If the outcome of the j-th request of f_i is Late-Warm, $\mathcal{B}_{i,j}$ is always t_e and $t_e < \text{OPT}_{i,j} < t_e + t_c$. Thus, for any outcome, $\mathcal{B}_{i,j}$ is always less than or equal to $\text{OPT}_{i,j}$, for any i, j. We define $\mathcal{B}(t_c)$ as the total latency of algorithm $\mathcal{B}$. According to the above definition of $\mathcal{B}$, $\mathcal{B}(t_c) \leq \text{OPT}(t_c)$. Besides, we find $\mathcal{B}$ computes the total latency in the same way as algorithms in the model without Late-Wram. The model with Late-Warm and the model without Late-Warm only differ in the method

of how they compute the total latency. So the solution of $\mathcal{B}$ is a feasible solution in the model without Late-Warm. Then, as $\mathrm{MOPT}(t_c)$ is the total latency of the offline optimal solution of online container caching without Late-Warm, $\mathrm{MOPT}(t_c) \leq \mathcal{B}(t_c)$. Thus, $\mathrm{MOPT}(t_c) \leq \mathrm{OPT}(t_c)$. Thus, $\mathrm{ALG}(t_c) \leq \mathrm{MALG}(t_c^2) \leq c \cdot \mathrm{MOPT}(t_c^2) \leq T_c \cdot c \cdot \mathrm{MOPT}(t_c) \leq T_c \cdot c \cdot \mathrm{OPT}(t_c)$. *i.e.*, $\mathrm{ALG}(t_c) \leq T_c \cdot c \cdot \mathrm{OPT}(t_c)$. Since the competitive ratio of OnCoLa on the model without Late-Warm is $O(K)$, OnCoLa is $O(T_c K)$-competitive for container caching with Late-Warm. □

V. Evaluation

We evaluate the performance of OnCoLa using the AliFC Trace [7], and the Azure Trace [21]. Compared with GD, the state-of-the-art algorithm that deals with container caching in serverless computing, OnCoLa can reduce the latency by up to 23.33%. Compared with LLB, the algorithm that supports relaying requests to other servers, it improves by 22.48%, under the default setting. Through sensitivity analysis on the number of edge servers, total memory size, initial memory threshold, γ, and relay latency, OnCoLa consistently outperforms baselines.

A. Methodology

Metrics. In this section, the metrics used to evaluate the performance of algorithms is the total latency incurred of all requests, including the execution time and cold start latency.

Memory Size. The total memory size of edge servers in OnCoLa is the sum of the sizes of containers corresponding to the most active functions, similar to [36]. The default configuration consists of 200 edge servers, and their total memory size is calculated as the sum of the initialized memory footprint of the top 40% active functions' containers. By default, γ is 0.6, relay latency is 200 ms, and the memory threshold is 60% for each server. To handle varying memory sizes among edge servers, we allocate the total memory size to N edge servers using Eqn. 3. Edge servers are divided into 5 types numbered i, where servers with the same $i\%N$ have the same memory size ($N = 200$).

$$K_i = (i\%N + 1)\left\lceil \frac{\text{Total Memory Size}}{(15\lfloor (N/5) \rfloor + \frac{(1+N\%5)(N\%5)}{2})} \right\rceil. \quad (3)$$

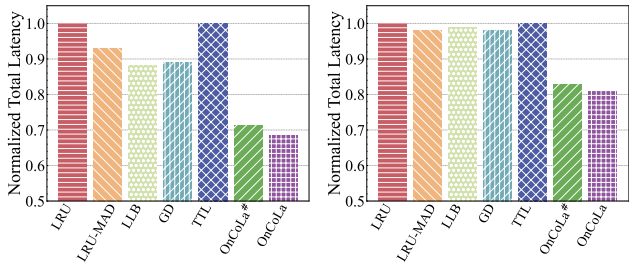
Workloads. Since both AliFC trace and Azure trace originate from serverless computing and lack edge server information for the requests, we use the Machine ID from Google's trace [37], and use Machine ID modular N as the edge server. The two traces differ in average request locality and average Late-Warm intensity [1]. With 200 servers, the AliFC trace has an average request locality of 0.161 and a Late-Warm intensity of 0.159, while the Azure trace has a request locality of 0.096 and a Late-Warm intensity of 0.485.

Baselines. We compare the performance of OnCoLa with LRU [38], TTL [39], LRU-MAD [36], GD [30] and LLB [40]. To verify the effectiveness of OnCoLa in handling relay latency, we also include OnCoLa# [1] as a baseline.[5]

[5] OnCoLa# does not take the relay latency into account, when c_f does not exist on s, if there is an edge server s' that has c_f and c_f with the lowest priority on s, r is relayed to s' [1].

B. Experiment Results

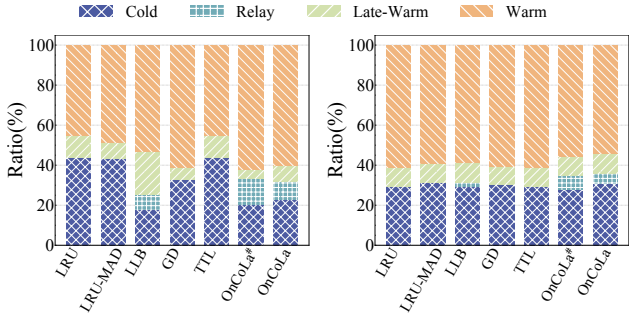
Overall performance. We evaluate OnCoLa's overall performance under default settings and compare it to baselines. Experimental results appear in Fig. 7, where the total latency of each algorithm is normalized to LRU (= 1.0). The experiments in AliFC show that among all baselines except OnCoLa#, LLB performs the best, with its improvement attributed to relaying requests to other servers. Compared to LLB, OnCoLa improves performance by 22.48%, employing priorities better suited for multi-server environments with Late-Warm. Furthermore, compared to GD, OnCoLa achieves a 23.33% latency reduction by leveraging inter-server cooperation and priorities tailored for serverless edge computing. Compared to OnCoLa#, OnCoLa utilizes its admission policy to handle relay latency, reducing latency by 4.21% in the AliFC trace and 2.41% in the Azure trace, respectively, under default settings.



(a) AliFC (b) Azure

Fig. 7. Overall performance.

Ingredient of Latency. Fig. 8 shows the ratios of Cold Start, Relay, Late-Warm, and Warm Start for each algorithm. Compared to OnCoLa#, OnCoLa demonstrates a reduction of 33.5% and 32.55% in Relay Ratio in the AliFC and Azure traces, respectively. This is attributed to OnCoLa's consideration of the relationship between relay latency and cold latency in Alg. 3. When relay latency approaches cold latency, OnCoLa may opt to initialize a container at the local server instead of relaying the request to another server with relay latency, a factor not considered by OnCoLa#.



(a) AliFC (b) Azure

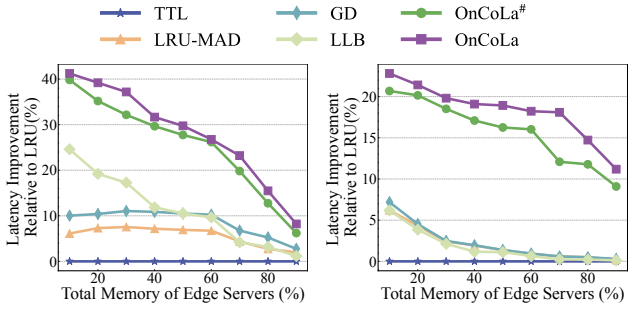
Fig. 8. Ratio of Cold-Start, Relay, Late-Warm, and Warm.

C. Sensitivity Analysis

We use the latency improvement relative to LRU to measure the performance of the algorithm, a higher latency improvement means better performance.

$$\text{Latency Improvement of A} = \frac{\text{Latency(LRU)} - \text{Latency(A)}}{\text{Latency(LRU)}}.$$

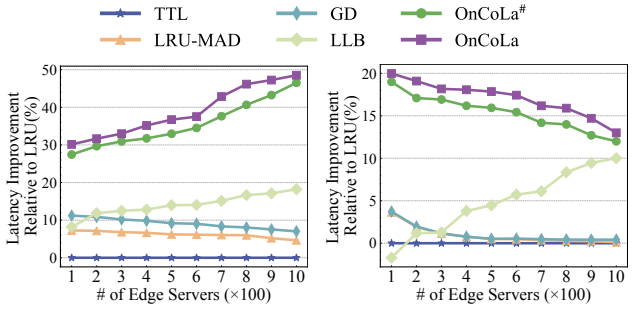
Total Memory Size. To investigate the impact of the total memory size, we vary it from 10% to 90% and display the results in Fig. 9. When the total memory size is small, OnCoLa exhibits higher improvement than other baselines. However, with sufficient memory to cache frequent containers, all algorithms' performances converged, especially in the AliFC trace. Compared to OnCoLa#, OnCoLa improves performance by 0.79% to 7.4% in the AliFC trace, and by 1.58% to 6.83% in the Azure trace.



(a) AliFC (b) Azure

Fig. 9. Impact of total memory size.

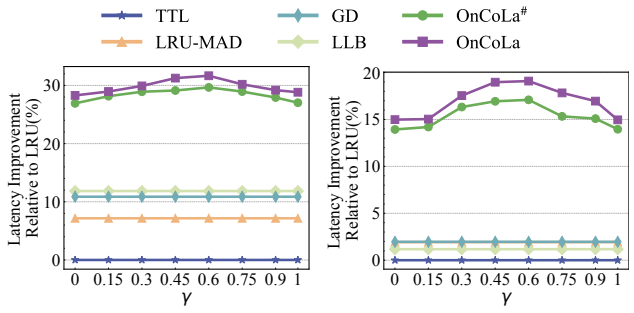
Number of Edge Servers. Fig.10 illustrates the impact of varying the number of edge servers. OnCoLa consistently demonstrates superior performance across a range of server counts, from 100 to 1000. In this experiment, compared to OnCoLa#, OnCoLa's performance improvement ranged from 1.14% to 9.27% across the two traces. With a fixed total memory size, increasing the number of servers results in less memory per server. In the AliFC trace, OnCoLa leverages request relaying among multiple servers, enhancing performance as server numbers increase. Conversely, in the Azure trace, performance declines due to the reduced per-server memory capacity, which limits container caching capabilities.



(a) AliFC (b) Azure

Fig. 10. Impact of the number of edge servers.

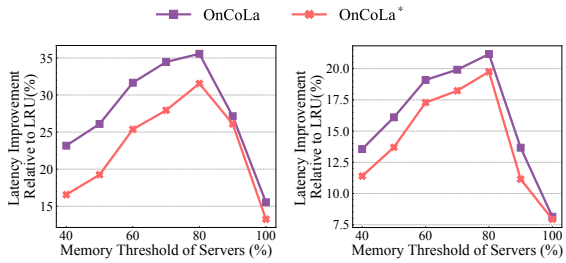
Parameter γ. We vary γ from 0 to 1, as shown in Fig. 11. The performance of OnCoLa fluctuates with changes in γ across both traces, peaking at $\gamma = 0.6$. In comparison, the variation of γ has a more pronounced effect in the Azure trace than in the AliFC trace. This is because γ adjusts the estimation of latency reduction during online execution from a cold start. The smaller impact of γ in the AliFC trace is reasonable, as the average Late-Warm intensity is lower than that in the Azure trace. Changes in γ affect the priority (p_c^f) used in the admission policy. As γ varies from 0 to 1, in the Azure trace, OnCoLa achieves up to 2.96% performance improvement over OnCoLa#.



(a) AliFC (b) Azure

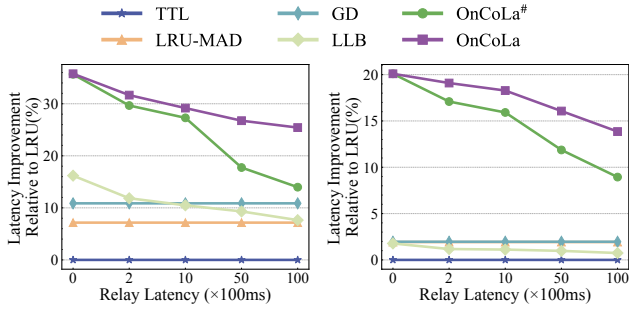
Fig. 11. Impact of γ.

Memory Adjustment. To assess the effectiveness of the memory adjustment method and demonstrate the impact of varying initial memory threshold, we vary threshold from 40% to 100%. We also assess the impact of memory growth by comparing OnCoLa with OnCoLa*, which lacks the memory growth feature. This evaluation involved setting LRU's memory threshold within the same range. As shown in Fig.12, OnCoLa's performance starts to decline when the threshold exceeds 80%, indicating that limiting the initial memory threshold is effective. Fig.12(a) shows that the impact of memory growth is more pronounced in the AliFC trace due to its higher average request locality compared to the Azure trace, making memory growth more likely to occur.



(a) AliFC (b) Azure

Fig. 12. Impact of memory adjustment.



(a) AliFC (b) Azure

Fig. 13. Impact of relay latency.

Relay Latency. To evaluate the effectiveness of the admission policy and explore the impact of varying relay latency, we adjust the relay latency from 0 to 10000 ms. Since LRU, LRU-MAD, and GD do not involve relaying requests to other servers, their performance remains constant. For OnCoLa, OnCoLa#, and LLB, the performance gain from relaying requests decreases as relay latency increases. When relay latency is 0 ms, OnCoLa achieves the same performance

as OnCoLa#. As relay latency increases from 200 to 10000 ms, OnCoLa demonstrates performance improvements over OnCoLa#. Specifically, for the AliFC trace, the improvements are 4.21%, 2.58%, 10.94%, and 13.31%, while for the Azure trace, the gains are 2.41%, 2.81%, 4.77%, and 5.4%. Due to the relay latency consideration of OnCoLa, it outperforms OnCoLa#, particularly at larger relay latencies.

VI. Implementation on Real Edge Devices

In this section, we evaluate OnCoLa on an edge cluster with PI4B and Nano, using three workloads of 10 common IoT data processing functions. OnCoLa reduces latency by 21.38% and cuts the request failure rate by up to 2.3× compared to the fixed-duration container caching policy.

A. Experimental Setup

Device and Platform. Our experiments are conducted on an edge cluster of 4 PI4B and 4 Nano, with another PI4B as the Relayer. Each PI4B has 1GB RAM, and each Nano has 4GB RAM, and a GPU. By default, the swap is turned off on all devices. We deploy OpenFaaS on Nano and faasd on PI4B, both using containerd as the container runtime. We enable GPU supported in OpenFaaS by mounting the nvidia-container-runtime on Nano [41]. The average relay latency among devices is 16.96 milliseconds.

Functions. In this experiment, as shown in Table II, we use 10 commonly used functions for IoT data processing as the composition of the workload. Due to the length constraints, Table II only displays the information on Nano. Moreover, terminating executing containers is not permitted, if all edge devices' memory is used by executing containers, requests for new functions will be marked as failed requests due to insufficient memory.

TABLE II
Functions used in experiment

f	t_f^c	t_f^e	z_f^p	z_f^e	Description
MM	2.13	1.53	25	104	Matrix Multiplication.
FFT	1.62	0.67	25	78	Fast Fourier Transform.
STT	1.32	1.12	22	58	Speech to Text.
AD	0.83	0.63	10	63	Audio Denoising.
RSA	1.66	1.34	11	58	Data Encryption.
PCA	1.23	2.81	13	72	Dimensionality Reduction.
RE	1.35	1.81	13	56	Resizing images.
IC	2.75	6.45	10	1060	Image Classification.
Node	1.32	0.03	19	21	OpenFaaS function.
Curl	1.41	0.20	6	8	OpenFaaS function.

f: function, t_f^c: cold start latency (s), t_f^e: execution time (s), z_f^p: initialized memory footprint (MB), z_f^e: executing memory footprint (MB). t_f^c and t_f^e are under 60% memory usage.

Workload. To evaluate the performance of OnCoLa under different workload types, we generate 3 types of workloads, with each workload containing 80,000 function requests across the 10 functions and 8 devices. The IC function requests cannot be processed on PI4B. The three workload types are:

- *Low*: 80% of requests are for 2 functions with small memory footprints and short execution times (Node and Curl), while the other 20% of requests are for the other functions. The average inter-request interval is 0.5 seconds.
- *Medium*: Each function has 8000 requests, which are evenly distributed to the 8 devices (except for IC). The average inter-request interval is 0.5 seconds.
- *High*: The number of function requests is the same as *Medium*. The average inter-request interval is 0.2 seconds.

B. Experimental Results

Average Latency. We compare OnCoLa ($\gamma = 0.6$, the memory threshold for PI4B and Nano is 40% and 60%, respectively) with the widely used TTL policy (which caches the containers for 5 minutes). And the metric is the average latency of all successfully completed requests. As Fig. 14 shown, OnCoLa reduces the latency by 10.16%, 21.38% and 14.75% for Low, Medium and High workloads, respectively.

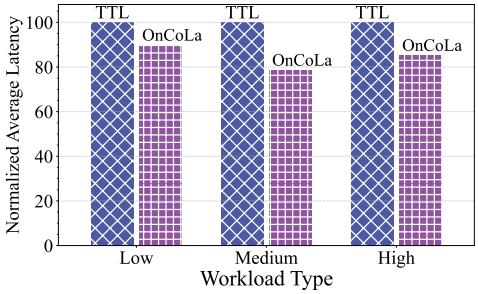
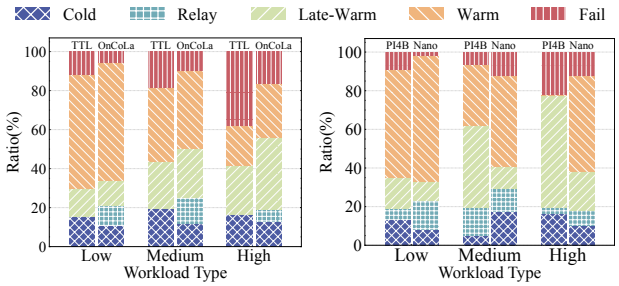


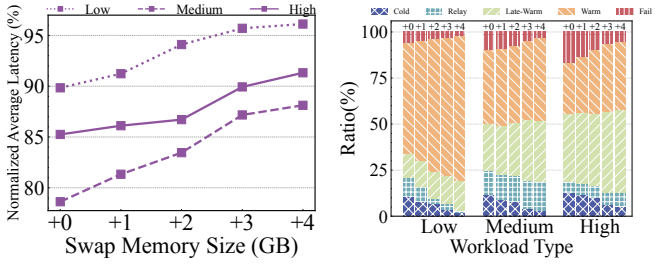
Fig. 14. The normalized average latency under three different workloads.

The Ratio of Outcomes. Fig. 15(a) compares the ratios of Cold, Relay, Late-Warm, Warm, and Fail for OnCoLa and TTL across three workloads. Fail, indicating the percentage of failed requests, rises from Low to High workload for both methods. OnCoLa reduces the failure rate by up to 2.3× compared to TTL. Under the Low workload, both OnCoLa and TTL exhibit a higher Warm Start ratio. In the High workload, with an average inter-request interval of 0.2s and more functions with longer cold start latency, Late-Warm ratios increase by 184.62% for OnCoLa and 76.92% for TTL, compared to the Low workload.



(a) Different type workloads (b) Different devices

Fig. 15. Ratio of outcomes for processing requests.



(a) Latency (b) Ratio of Outcomes

Fig. 16. Impact of memory size of devices.

The Ratio of Outcomes on Different Devices. Fig. 15(b) shows the outcome ratios on PI4B and Nano with OnCoLa.

For the Low workload, Warm ratios are high at 55.77% (PI4B) and 65.12% (Nano). Under the Medium workload, Nano's Fail ratio rises to 12% due to increased IC requests. In the High workload, Fail reaches 22%, Late-Warm grows to 57.78%, and Warm on PI4B drops to 0.44%.

Impact of Memory Size of Devices. To assess `OnCoLa`'s scalability with memory size, we enabled swap to extend PI4B and Nano memory from 1GB to 4GB. Results in Fig. 16 show that as swap memory increases, `OnCoLa`'s performance decreases slightly but stays at least 3.88% better than TTL. Fig. 16(b) highlights that increased swap memory raises the Warm proportion while reducing Fail rates.

VII. Related Works

A. Serverless Edge Computing

Serverless edge computing, merges the benefits of serverless computing, such as on-demand resource allocation and fine-grained resource management, with edge computing's proximity to data sources, offering reduced latency for applications like the Internet of Things. However, the resource-limited and heterogeneous nature of the edge introduces challenges not typically found in cloud-based serverless environments. Research in serverless edge computing aims to optimize performance and resource utilization, various studies focus on deployment [42], pricing [43], [44], scheduling [22]–[25], resource provisioning [45], cold start mitigation [46] and handling function dependencies [47]. While these works address many aspects of serverless edge computing, this work focuses on the online container caching problem for IoT data processing within serverless edge computing. We introduce a novel consideration of Late-Warm, alongside the practical challenges of memory sensitivity on resource-limited devices and request relaying between edge servers. Our proposed algorithm, `OnCoLa`, aims to minimize total latency by tackling these challenges inherent to the serverless edge environment.

B. Container Caching

Major serverless platforms like AWS and Azure use a fixed duration caching policy [48]. FaaSCache [30] uses a Greedy-Dual keep-alive policy considering the request frequency and function patterns. Shahrad *et al.* [21] propose a practical resource management policy for container caching and pre-warming. Yu *et al.* [19] propose a layer-wise container pre-warming and keep-alive policy to reduce cold starts. Flame is a cache system with centralized control for optimized cache-hit ratio and resource efficiency across clusters [49]. In this paper, we focus on container caching in serverless edge computing, which faces challenges from Memory Sensitivity, Request Relaying and Late-Warm. Existing online caching algorithms typically assume constant latency and memory footprint, and container caching policies for powerful servers often neglect edge-specific constraints. We introduce `OnCoLa`, a novel approach that assigns a priority to each container, effectively dealing with these three challenges all at once with the competitive ratio.

VIII. Conclusion

This paper investigates the online container caching problem in serverless edge computing. We highlight the new challenges of designing an online caching algorithm with resource-limited edge servers, including Late-Warm, Memory-sensitivity, and Request Relaying. We propose an $O(T_c K)$-competitive algorithm, `OnCoLa`, to address these challenges. We implement `OnCoLa` and conduct experiments on edge devices to validate the improvement over the current policy. We conduct extensive simulations based on real-world traces and show that `OnCoLa` outperforms baselines. Serverless edge computing is still in its early stages, we hope this work can contribute to implementing the serverless paradigm in large-scale edge systems. The source code is available for reference.[6]

References

[1] G. Li, H. Tan, X. Zhang, C. Zhang, R. Zhou, Z. Han, and G. Chen, "Online container caching with late-warm for iot data processing," in *IEEE ICDE 2024*, 2024, pp. 1547–1560.

[2] M. Adil, M. Attique, M. M. Jadoon, J. Ali, A. Farouk, and H. Song, "Hopctp: a robust channel categorization data preservation scheme for industrial healthcare internet of things," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 10, pp. 7151–7161, 2022.

[3] P. Killeen, I. Kiringa, and T. Yeap, "Unsupervised dynamic sensor selection for iot-based predictive maintenance of a fleet of public transport buses," *ACM Transactions on Internet of Things*, vol. 3, no. 3, pp. 1–36, 2022.

[4] J. E. Tate, "Preprocessing and golomb–rice encoding for lossless compression of phasor angle data," *IEEE transactions on smart grid*, vol. 7, no. 2, pp. 718–729, 2015.

[5] M. Mazaheri, R. Ruiz, D. Giustiniano, J. Widmer, and O. Abari, "Bringing millimeter wave technology to any iot device," in *ACM MobiCom 2023*, pp. 1–15.

[6] D. Carrizales-Espinoza, D. D. Sanchez-Gallegos, J. Gonzalez-Compean, and J. Carretero, "Structmesh: A storage framework for serverless computing continuum," *Future Generation Computer Systems*, vol. 159, pp. 353–369, 2024.

[7] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *USENIX ATC 2021*.

[8] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[9] B. Javadi, B. Jingtao Sun, and R. Ranjan, "Serverless architecture for edge computing." IET, 2020.

[10] Z. Wen, Q. Chen, Q. Deng, Y. Niu, Z. Song, and F. Liu, "Combo-func: Joint resource combination and container placement for serverless function scaling with heterogeneous container," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 11, pp. 1989 – 2005, 2024.

[11] S. Kohli, S. Kharbanda, R. Bruno, J. Carreira, and P. Fonseca, "Pronghorn: Effective checkpoint orchestration for serverless hot-starts," in *EuroSys 2024*, pp. 298–316.

[12] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "Serverlessllm: Low-latency serverless inference for large language models," in *USENIX OSDI 2024*, pp. 135–153.

[13] A. Szekely, A. Belay, R. Morris, and M. F. Kaashoek, "Unifying serverless and microservice workloads with sigmaos," in *ACM SOSP 2024*, pp. 385–402.

[14] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. Darlow, J. Wang, Q. Deng, and A. Barker, "Serverless cold starts and where to find them," in *EuroSys 2025*, p. 938–953.

[15] Z. Zhang, C. Jin, and X. Jin, "Jolteon: Unleashing the promise of serverless for serverless workflows," in *USENIX NSDI 2024*.

[16] Q. Chen, J. Qian, Y. Che, Z. Lin, J. Wang, J. Zhou, L. Song, Y. Liang, J. Wu, W. Zheng *et al.*, "Yuanrong: A production general-purpose serverless system for distributed applications in the cloud," in *ACM SIGCOMM 2024*, pp. 843–859.

[6] https://drive.google.com/drive/folders/1Fh5IDVyu66VzQUuiWwfxjkLsZQZGAyZH

[17] X. Yue, S. Yang, L. Zhu, S. Trajanovski, F. Li, and X. Fu, "Exploiting wide-area resource elasticity with fine-grained orchestration for serverless analytics," *IEEE/ACM Transactions on Networking*, 2024.
[18] Q. Liu, Y. Cheng, H. Shen, A. Wang, and B. Balaji, "Concurrency-informed orchestration for serverless functions," in *ACM ASPLOS 2025*.
[19] H. Yu, R. Basu Roy, C. Fontenot, D. Tiwari, J. Li, H. Zhang, H. Wang, and S.-J. Park, "Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing," in *ACM ASPLOS 2024*.
[20] J. Stojkovic, N. Iliakopoulou, T. Xu, H. Franke, and J. Torrellas, "Ecofaas: Rethinking the design of serverless environments for energy efficiency," in *ACM/IEEE ISCA 2024*.
[21] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX ATC 2020*.
[22] X. Shang, Y. Mao, Y. Liu, Y. Huang, Z. Liu, and Y. Yang, "Online container scheduling for data-intensive applications in serverless edge computing," in *IEEE INFOCOM 2023*, pp. 1–10.
[23] M. S. Aslanpour, A. N. Toosi, M. A. Cheema, and M. B. Chhetri, "Faashouse: sustainable serverless edge computing through energy-aware resource scheduling," *IEEE Transactions on Services Computing*, vol. 17, no. 4, pp. 1533–1547, 2024.
[24] F. Tütüncüoğlu, S. Jošilo, and G. Dán, "Online learning for rate-adaptive task offloading under latency constraints in serverless edge computing," *IEEE/ACM Transactions on Networking*, vol. 31, no. 2, 2022.
[25] S. Hu, Z. Qu, B. Tang, B. Ye, G. Li, and W. Shi, "Joint service request scheduling and container retention in serverless edge computing for vehicle-infrastructure collaboration," *IEEE Transactions on Mobile Computing*, vol. 23, no. 6, pp. 6508–6521, 2023.
[26] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.
[27] N. E. Young, "On-line file caching," *Algorithmica*, vol. 33, no. 3, pp. 371–383, 2002.
[28] H. Tan, S. H.-C. Jiang, Z. Han, L. Liu, K. Han, and Q. Zhao, "Camul: Online caching on multiple caches with relaying and bypassing," in *IEEE INFOCOM 2019*.
[29] D. Rohatgi, "Near-optimal bounds for online caching with machine learned advice," in *SIAM SODA 2020*.
[30] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *ACM ASPLOS 2021*.
[31] "Use containers to build, share and run your applications," 2024, https://www.docker.com/resources/what-container/.
[32] P. Manohar and J. Williams, "Lower bounds for caching with delayed hits," *arXiv preprint arXiv:2006.00376*, 2020.
[33] "of-watchdog," 2024, https://github.com/openfaas/of-watchdog.
[34] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache." in *USENIX FAST 2003*.
[35] S. Albers, "Brics, mini-course on competitive online algorithms," *Aarhus University*, vol. 32, 1996.
[36] N. Atre, J. Sherry, W. Wang, and D. S. Berger, "Caching with delayed hits," in *ACM SIGCOMM 2020*.
[37] C. Reiss, J. Wilkes, and J. Hellerstein, "Google cluster-usage trace," in *Technical Report*, 2011.
[38] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Communications of the ACM*, vol. 28, no. 2, pp. 202–208, 1985.
[39] "Apache openwhisk," 2024, https://openwhisk.apache.org.
[40] L. Epstein, C. Imreh, A. Levin, and J. Nagy-György, "Online file caching with rejection penalties," *Algorithmica*, vol. 71, pp. 279–306, 2015.
[41] "Nvidia container runtime," 2024, https://developer.nvidia.com/nvidia-container-runtime.
[42] K. Cao, M. Chen, S. Karnouskos, and S. Hu, "Reliability-aware personalized deployment of approximate computation iot applications in serverless mobile edge computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.
[43] F. Tütüncüoğlu, A. Ben-Ameur, G. Dán, A. Araldo, and T. Chahed, "Dynamic time-of-use pricing for serverless edge computing with generalized hidden parameter markov decision processes," in *IEEE ICDCS 2024*, pp. 668–679.
[44] F. Tütüncüoğlu and G. Dán, "Joint resource management and pricing for task offloading in serverless edge computing," *IEEE Transactions on Mobile Computing*, vol. 23, no. 6, pp. 7438–7452, 2023.
[45] O. Ascigil, A. G. Tasiopoulos, T. K. Phan, V. Sourlas, I. Psaras, and G. Pavlou, "Resource provisioning and allocation in function-as-a-service edge-clouds," *IEEE Transactions on Services Computing*, vol. 15, no. 4, pp. 2410–2424, 2021.
[46] K. Zhao, Z. Zhou, L. Jiao, S. Cai, F. Xu, and X. Chen, "Taming serverless cold start of cloud model inference with edge computing," *IEEE Transactions on Mobile Computing*, vol. 23, no. 8, 2023.
[47] S. Deng, H. Zhao, Z. Xiang, C. Zhang, R. Jiang, Y. Li, J. Yin, S. Dustdar, and A. Y. Zomaya, "Dependent function embedding for distributed serverless edge computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2346–2357, 2021.
[48] "Aws lambda," 2024, https://aws.amazon.com/lambda.
[49] Y. Yang, L. Zhao, Y. Li, S. Wu, Y. Hao, Y. Ma, and K. Li, "Flame: A centralized cache controller for serverless computing," in *ACM ASPLOS 2023*, pp. 153–168.

Guopeng Li received his B.Eng. degree in Computer Science and Technology from Central South University in 2020. He is currently pursuing his Ph.D degree at University of Science and Technology of China (USTC). His main research interests include serverless computing, edge intelligence, LLM-based applications, and efficient LLM systems.

Haisheng Tan (Senior Member, IEEE) received his B.E. degree in Software Engineering and B.S. degree in Management both from University of Science and Technology of China (USTC) with the highest honor. Then, he got his Ph.D. degree in computer science at the University of Hong Kong (HKU). He is currently a professor at USTC. His research interests include algorithms and networking. Dr. Tan has published over 80 papers in prestigious journals and conferences, mainly in the areas of AIoT and edge computing. He recently received the Best Paper Award in WASA'19, CWSN'20, PDCAT'20, and ICAPDS'21.

Chi Zhang received his B.Eng. degree in Computer Science and Technology from USTC with the honor of The Talent Program in Computer and Information Science and Technology in 2017, and got his Ph.D. degree in Computer Science and Technology from USTC in 2023. He is currently an associate professor at Hefei University of Technology. His primary research interests are cloud computing and algorithms.

Xuan Zhang received her B.Eng. degree in Computer Science and Technology from Northwest University, China, in 2023. She is currently pursuing her MS degree at University of Science and Technology of China (USTC). Her main research interest is edge computing and system for AI.

Zhenhua Han is a senior researcher at Microsoft Research (Asia), Shanghai. He received B.Eng. degree in electronic and information engineering from University of Electronic Science and Technology of China, Chengdu, in 2014, and Ph.D. degree from the University of Hong Kong, Hong Kong. His research interests are resource management, systems for machine learning, and cloud computing. Many of his works have been published in top venues such as USENIX OSDI, ACM SOSP, and ASPLOS.

Guoliang Chen received the B.S. degree from Xi'an Jiaotong University, Xi'an, in 1961. Since 1973, he has been with the University of Science and Technology of China, Hefei, a professor in computer science. From 1981 to 1983, he was a visiting scholar at Purdue University, West Lafayette, IN. He has published 9 books and more than 200 research papers. His research interests include parallel algorithms, computer architectures, computer networks, and computational intelligence. He is an academician of the Chinese Academy of Sciences.