Cross-Edge Orchestration of Serverless Functions with Probabilistic Caching

Chen Chen*, Manuel Herrera§, Ge Zheng§, Liqiao Xia†§, Zhengyang Ling§, Jiangtao Wang‡
*Department of Computer Science and Technology, University of Cambridge, UK
§Department of Engineering, University of Cambridge, UK
†Department of Industrial and Systems Engineering, Hong Kong Polytechnic University, China
‡Research Centre for Intelligent Healthcare, Coventry University, UK
Email: cc2181@cam.ac.uk, amh226@cam.ac.uk, gz305@cam.ac.uk, liqiao.xia@connect.polyu.hk, zl461@cam.ac.uk, jiangtao.wang@coventry.ac.uk

***Abstract*— Serverless edge computing adopts an event-based paradigm that provides back-end services and dynamically provisions resources as needed, resulting in efficient resource utilization. To improve the end-to-end latency and revenue, service providers need to optimize the number and placement of serverless containers while considering the system cost (i.e., latency cost and container running cost) incurred by the provisioning. The particular reason for this circumstance is that frequently creating and destroying containers not only increases the system cost but also degrades the time responsiveness due to the cold-start process. Function caching is a common approach to mitigate the cold-start issue. However, function caching requires extra hardware resources and hence incurs extra system costs. Furthermore, the dynamic and bursty nature of serverless invocations remains an under-explored area. Hence, it is vitally important for service providers to conduct a context-aware request distribution and container caching policy for serverless edge computing. In this paper, we study the request distribution and container caching problem in serverless edge computing. We prove the proposed problem is NP-hard and hence difficult to find a global optimal solution. We jointly consider the distributed and resource-constrained nature of edge computing and propose an optimized request distribution algorithm that adapts to the dynamics of serverless invocations with a theoretical performance guarantee. Also, we propose a context-aware probabilistic caching policy that incorporates a number of characteristics of serverless invocations. Via simulation and implementation results, we demonstrate the superiority of the proposed algorithm by outperforming existing caching policies in terms of the overall system cost and cold-start frequency by up to 62.1% and 69.1%, respectively.**

***Keywords*—Edge Computing, Serverless Computing, Resource Allocation**

I. Introduction

With the accelerated penetration of 5G communications and Internet-of-Things (IoTs), a wide range of mobile and IoT applications are expected to be connected to the Internet, ranging from smart factories to edge computing-assisted online gaming, augmented reality and virtual reality [1], [2]. As a result of deploying these diverse applications, a large amount of multi-modal data (e.g., video, image and audio) of the physical environment is continuously collected at various edge devices [3], [4]. To process such a tremendous amount of data in a time-responsive manner, edge computing has been introduced as a promising approach. Edge computing, with a distributed nature, pushes computing resources and services from the network core to the network edges that are located in the proximity of mobile users and IoT applications, resulting in a significant reduction of end-to-end latency [5], [6].

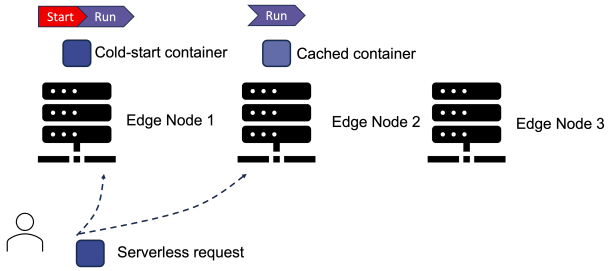


Fig. 1: Function caching in serverless edge computing

Recently, serverless computing has introduced new inspirations to edge computing and receives significant attention [7], [8]. Serverless computing is a service paradigm that provides simplified deployment, elastic computing, and fast responsiveness in an event-driven model [9], [10]. Serverless paradigm encapsulates IoT services in lightweight containers and only deploys them when invoked by events requested by services [11], [12]. The containers are destroyed after completing the tasks and the hardware resources are released, providing resource efficiency and fast response simultaneously [13]. For example, Microsoft Azure provides a comprehensive family of most advanced AI services, such as GPT-3.5, ChatGPT, DALL·E, to enable intelligent, cutting-edge and low-latency applications [14].

As illustrated in Figure 1, when end-user requests a serverless function, distributing containers near the data origin (edge node 1) may lead to significant delay due to the cold-start process of creating a container [15], [16], [17], [18]. Otherwise, the system can distribute the request to edge node 2, where a cached container may exist, to avoid cold-start issue at the cost of larger transmission delays. Hence, unlike centralized cloud datacenters with homogeneous computing and storage substrates, supporting serverless funcitons between distributed edge nodes may offset the benefits of serverless

paradigm.

When deploying various serverless containers across multiple geographically distributed edge nodes, fully optimizing the benefits of serverless containers is very challenging for the following reasons. First, pioneering work largely overlooks the heterogeneous network topology in edge computing. For instance, the origin of end-users may be distant from an edge node that holds the corresponding container. Thus, besides container scheduling, how to jointly incorporate the data origin and containers for low-transmission delay is non-trivial for an efficient serverless platform [19]. Second, compared to cloud computing with seemingly endless computing resources, edge devices may regularly run at close to capacity [20], [21] and memory resources are still expensive at edge clouds [8], [22]. Frequently creating and destroying containers significantly increases the system cost because creating containers requires downloading the code and starting a new execution environment before the requests can be served [23], [24]. Hence, optimizing container orchestration without statistical knowledge of future invocations is challenging. Although function caching can be used to mitigate the above issue, keeping a function alive is not free. Due to the pay-as-you-go model in serverless computing, users are not billed for functions that are not executing and hence the service providers need to undertake the cost of container caching. Hence, caching a number of containers in edge clouds massively increases the overall system cost. Function caching will deteriorate service providers' revenue in the long term. Therefore, balancing the container switching, communication and caching cost is pivotal to achieving an efficient system in edge clouds and this trade-off has not been explicitly studied so far.

Keeping the above factors in mind, in this work, we aim to optimize the system efficiency defined by the combination of service latency and resource utilization. We propose an online probabilistic caching policy to orchestrate serverless containers in edge computing, optimizing the system cost of the cross-edge serverless system. The proposed framework considers (1) the container switching cost that is proportional to the cold-start delay. (2) the communication cost incurred by the latency between edge nodes. (3) the container running cost incurred by using hardware resources. With the above setup, the cost minimization problem for cross-edge networks over the long term is formulated as integer linear programming (ILP) problem.

To optimize the unified cost, the proposed framework applies two policies: (1) A request distribution algorithm that dynamically assigns serverless requests to different edge nodes. This approach jointly considers the communication cost and the switching cost. (2) A context-aware probabilistic caching policy that captures the characteristics of serverless invocations. By doing so, it aims to improve the efficiency of caching in serverless edge computing environments. Our main contributions are:

- We formulate a request distribution and container caching problem as an Integer Linear Programming (ILP) problem that jointly considers communication cost, container switching cost, and container running cost. We prove the NP-hardness of the problem, showing the complexity of finding a global optimal solution. Then, we propose an online competitive algorithm for request distribution with a theoretical performance guarantee.
- We propose an online request distribution algorithm that adapts to the dynamics of incoming requests at each edge cloud. Due to the burstiness of serverless edge computing, workloads can vary significantly over time, and the ability to adjust caching strategies in response to these dynamics is pivotal.
- We propose a probabilistic caching policy (**pCache**) that captures the context of serverless functions, including the invocation frequency, memory usage and recency. This approach is novel because it goes beyond simple caching strategies and considers the specific characteristics of each invocation in making caching decisions. The awareness of serverless context sets the paper apart from existing caching policies.
- To evaluate the performance, we use real-world traces and topology. We conduct extensive experiments over simulations and a serverless platform Knative [25], showing the superiority of our algorithm compared with state-of-the-art caching policies.

The remainder of the paper is organized as follows. In Section II we overview the related works of serverless edge computing, including request distribution and caching. The state-of-the-art studies are summarized and the existing challenges are identified. Section III introduces the system model and the system design. In Section IV, we formulate the optimization problem and prove its NP-hardness. Section V presents our proposed algorithm, including the request distribution and the caching algorithms. We also prove the theoretical bound of the proposed algorithms, proving the efficiency of the proposed algorithms. Section VI evaluates the algorithm and conclusions are drawn in Section VII. Moreover, we describe the prospects of serverless edge computing from our point of view and come up with some research opportunities.

II. Related Work

A number of works investigate the function distribution problem in edge computing. Defuse [26] addresses the issue of performance degradation caused by cold-starts in serverless platforms. By constructing a function dependency graph, Defuse reduces the occurrences of cold-starts and demonstrates improved memory usage and decreased cold-start frequency compared to existing methods. S-Cache [27] investigates the problem of container placement with latency optimization in edge computing environments. A priority-based algorithm is proposed to determine container termination in serverless computing. The experimental results demonstrate that the proposed algorithm improves end-to-end response time. However, S-Cache does not consider the container running cost and aggressively caches containers with high priorities. Aslanpour *et al.* [28] focus on energy-aware scheduling in serverless edge computing. Zone-oriented and priority-based algorithms are proposed to improve the operational availability of bottleneck nodes, introducing concepts such as "sticky offloading" and "warm scheduling" for QoS

optimization. Many other works [29], [30], [5] explore the cold-start problem in serverless computing but overlook the characteristics of serverless invocations.

A wide range of research efforts have been carried out, focusing on service placement in edge computing. Poularakis *et al.* [31] propose a joint optimization problem of service placement and request routing in Mobile Edge Computing (MEC) networks with multidimensional constraints. The proposed algorithm achieves near-optimal performance by using randomized rounding. Gu *et al.* [18] focus on optimizing container-based microservice placement and request scheduling in edge computing. The potential of layer sharing among co-located microservices is adopted to enhance throughput and increase the number of hosted microservices. An iterative greedy algorithm is proposed with a guaranteed approximation ratio.

A few works also investigate the container scheduling problem. Lin *et al.* [32] investigate the challenges faced by cloud users when migrating applications in serverless computing. A heuristic algorithm called Probability Refined Critical Path Greedy algorithm (PRCP) is introduced to optimize the cost of serverless applications. The proposed models and algorithm are extensively evaluated on AWS Lambda and Step Functions, demonstrating their effectiveness. Rausch *et al.* [33] introduce a container scheduling system called Skippy that optimizes task placement on edge infrastructures by considering factors such as data and computation movement, GPU acceleration, and operational objectives. The results demonstrate the effectiveness of Skippy in improving task placement quality and achieving operational goals. Lin *et al.* [34] propose performance modeling and optimization algorithms by considering the cost in monetary terms. The proposed algorithms provide prediction of performance and cost, helping developers to make decisions. These research efforts do not consider the dynamic and bursty nature of serverless invocations and hence are not applicable to our proposed problem.

In the context of edge computing, content caching is widely used as it can significantly reduce the delay in content delivery. Tadrous *et al.* [35] formulate a problem of proactive caching in mobile data networks, aiming to minimize service costs and improve delivery efficiency. The proposed solutions show close-to-optimal performance even with small prediction windows. Jia *et al.* [36] formulate a reliability-aware Service Function Chain (SFC) scheduling problem in a 5G network environment. A mixed integer non-linear programming problem is formulated. Also, an efficient algorithm for determining Virtual Network Functions (VNF) redundancy and a reinforcement learning-based scheduling algorithm are proposed. Simulation results demonstrate the effectiveness of the proposed approach in increasing the success rate of dynamically arriving SFCs. Pan *et al.* [7] investigate the retention-aware container caching problem in serverless edge computing. An optimization approach is proposed by using container caching and request distribution to improve system efficiency. By mapping the problem to the ski-rental problem and developing online algorithms, the study demonstrates significant performance gains compared to existing caching strategies. Farhadi *et al.* [37] study the optimization of service placement and request scheduling in mobile edge computing environments. A two-time-scale framework is presented by jointly considering storage, communication, computation, and budget constraints. Extensive simulations demonstrate that the algorithm achieves near-optimal performance. Stephen *et al.* [38] formulate a placement problem in heterogeneous mobile edge computing. The problem is proven to be NP-hard and a deterministic approximation algorithm is proposed with performance guarantee. The algorithm utilizes novel slot constructions on edge nodes and applies the method of conditional expectations for approximation guarantees. Simulation results demonstrate the superiority of the proposed algorithm over existing approaches. Zhou *et al.* [39] aim to optimize the service latency in UAV-assisted wirless mobile networks by incorporating the unique features of UAV. To reduce the caching overhead, Lyapunov optimization approach and dependent rounding technique are adopted to achieve a near-optimal performance. Cao *et al.* [26] propose an optimal auction mechanism to decide the cache space allocation and user payments in edge computing.

However, these works focus on content caching policies which are not directly applicable to function caching. In function caching, one function may not serve multiple requests simultaneously while content caching can. Furthermore, our work considers the request distribution and connectivity between edge nodes which renders existing works for content caching inapplicable.

III. System Model

In this section, we present the system model for the serverless request distribution and caching problem. After that, we present the system design of the proposed framework.

A. Overview of the cross-edge system

In this paper, we consider an edge service provider deploying serverless computing services on a set of geographically distributed edge nodes, denoted by $\mathcal{V} = \{1, 2, ..., V\}$. Each edge node v is equipped with a certain amount of hardware resources (e.g., memory) illustrated as U_t^v. Similarly, we use $\mathcal{E} = \{1, 2, ..., E\}$ to represent the set of links between edge nodes. Also, the set of user requests is denoted by $\mathcal{R} = \{1, 2, ..., R\}$. The system works in a time-interval manner spanning across a large period of time $\mathcal{T}$ and each time slot is denoted by $t \in \mathcal{T}$. Each time interval denotes a decision interval, which is much longer than the processing time of a typical serverless application [40].

The service requests are generated by end-users. A central scheduler orchestrates the service requests to appropriate edge clouds that serve the requests.

B. System design

Figure 2 shows the architecture of pCache which is built over Knative, Kubernetes [41] and Kourier [42] tools. The users first send severless requests to an ingress gateway which is extended from Kourier gateway. After that, the serverless requests are sent to pCache's scheduler. The scheduler makes decisions based on the Algorithm 1 and 2. The cache information is also stored in the scheduler and hence the scheduler

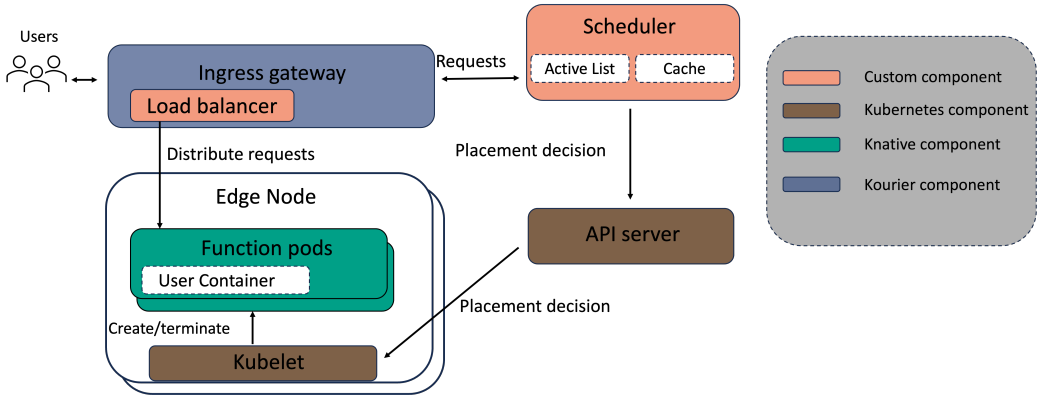


Fig. 2: pCache overview

jointly factors in the cached containers and the incoming requests. The decisions are then sent to the endpoints of Knative resources through the Kubernetes' API server. The API server will create new containers if required. In the meantime, the placement decisions are also sent to the ingress gateway and the load balancer will distribute the traffic to an assigned function pod. Eventually, the function pods process the incoming traffic and return results back to users.

It is worth mentioning that the scheduler keeps an active list and a cache. The active list contains information on active containers that are serving requests. The cache contains information on containers that are alive but not serving any requests. When a container finishes serving a request, the scheduler will decide whether to terminate the container based on the probabilistic algorithm 2.

IV. Problem Description

In this work, the system cost of the online serverless function orchestration includes two components: the service latency cost and the container running cost. All symbols and variables are listed in Table I.

A. Service latency cost

The service latency cost consists of two components: the switching cost and the communication cost which are all proportional to the actual latency.

Launching a new serverless container requires transferring the container image containing the serverless application to the hosting edge node and creating a new executive environment which may take up to a few seconds. We use p_v^n to denote the cost of creating a type n container at edge cloud v. Then, the total container switching cost at time interval t is given by:

$$\sum_{v\in\mathcal{V}}\sum_{n\in\mathcal{N}} p_n^v max\{m_{n,t}^v - a_{n,t}^v, 0\} \tag{1}$$

where $a_{n,t}^v = max\{a_{n,t-1}^v, m_{n,t-1}^v\} - y_{n,t-1}^v$. Let $a_{n,t}^v$ represent the number of active containers of type n on node v at the beginning of time interval t. $m_{n,t}^v$ denotes the number of required containers of type n on node v at the beginning

TABLE I: Symbols and Variables

Symbols and Variables	Description
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Physical network graph
$\mathcal{V}$	Set of edge nodes
$\mathcal{E}$	Set of links
$\mathcal{N}$	Set of container types
$\mathcal{T}$	Set of time intervals
u_n	The required hardware resource of type n container
U_t^v	The hardware capacity of node v
p_n^v	The cost of creating a type n container at node v
q_n^v	The running cost of type n container at node v
$d_{v,v'}$	Communication cost between node v and v'
$m_{n,t}^v$	Number of type n container assigned to node v
$m_{n,t}^{v->v'}$	Number of type n container generated at node v and assigned to node v'
$a_n^{v,t}$	Number of type n containers active at node v in time interval t
$\lambda_{n,t}^v$	The total number of requests generated at node v.
$y_{n,t-1}^v$	The number of containers to be destroyed on node v at end of $t-1$.
$k_{v,t}^n$	The number of requests remaining to be served
α	Parameter to tune the tradeoff between service latency and container running cost
β	Parameter to tune Zipf distribution

of time interval t. $y^v_{n,t-1}$ represents the number of containers to be destroyed on node v in the end of time interval $t-1$.

We use $d_{v,v'}$ to denote the communication cost between two edge nodes v and v' which is proportional to the network communication latency. The total network communication cost in time slot t is given by:

$$\sum_{v,v' \in \mathcal{V}} \sum_{n \in \mathcal{N}} d_{v,v'} m^{v->v'}_{n,t} \tag{2}$$

where $m^{v->v'}_{n,t}$ represents the number of containers that are generated at edge node v and distributed to edge node v'.

Hence, the total service latency cost is given as follows.

$$C_L(t) = \sum_{v \in \mathcal{V}} \sum_{n \in \mathcal{N}} p^v_n max\{m^v_{n,t} - a^v_{n,t}, 0\} + \sum_{v,v' \in \mathcal{V}} \sum_{n \in \mathcal{N}} d_{v,v'} m^{v->v'}_{n,t} \tag{3}$$

B. Container running cost

The container running cost denotes the hardware resource price paid for running a container. Hence, the total container running cost in time interval t is given as follows.

$$C_R(t) = \sum_{v \in \mathcal{V}} \sum_{n \in \mathcal{N}} q^v_n m^v_{n,t} \tag{4}$$

where $q^v_n(t)$ represents the system cost of running a container n in edge cloud v in one time slot, attributed to hardware resources paid for running a container.

Hence, the total cost of the system is a sum of the service latency cost and the container running cost.

$$C_L(t) + \alpha C_R(t) \tag{5}$$

where the parameter α is used to tune the tradeoff between service latency cost and container running cost. Note that $\alpha q^v_n \leq p^v_n$ should always be followed, otherwise caching containers will be more expensive than creating new containers, making the function caching unhelpful.

C. The system cost minimization problem

In this work, we aim to devise a cost-efficient request distribution and caching framework for serverless edge computing. To this end, we formulate a joint optimization on container placement and container caching, aiming at minimizing the overall system cost.

***Problem** 1:*

$$min \quad \sum_{t \in \mathcal{T}} (C_L(t) + \alpha C_R(t)) \tag{6}$$

s.t.

$$\sum_{n \in \mathcal{N}} u_n m^v_{n,t} \leq U^v_t, \forall v \in \mathcal{V}, \forall t \in \mathcal{T} \tag{7}$$

$$\sum_{n \in \mathcal{N}} m^{v->v'}_{n,t} = \lambda^v_{n,t}, \forall v, v' \in \mathcal{V}, \forall t \in \mathcal{T} \tag{8}$$

$$y^v_n(t) \in [0, max\{a^{n,t}_v, m^{n,t}_v\}] \quad \forall t \in \mathcal{T}, \forall n \in \mathcal{N}, v \in \mathcal{V} \tag{9}$$

$$m^v_{n,t} \in \{0, 1, ..., \lambda^v_n\}, \quad \forall t \in \mathcal{T}, \forall n \in \mathcal{N}, v \in \mathcal{V} \tag{10}$$

Eq. 7 guarantees that the allocated hardware resource does not exceed the maximum capacity on each node. Eq. 8 ensures that every request is distributed to only one node. Eq. 9 ensures that the number of destroyed containers is less than the number of active containers. Eq. 10 represents the integrality constraint for the number of distributed containers $m^v_{n,t}$.

D. NP hardness

We show that the Generalized Assignment Problem (GAP) [43], which is proven to be NP-hard, can be reduced to a simplified version of the proposed problem. The GAP problem is assigning a number of K jobs to a set of J agents, aiming to minimize the total cost. Let ω^k_j denote the size of job k for agent j to perform the job. Let d^k_j represent the cost of running job k for agent j. Also, we use Ω^k_j to denote the capacity of agent j and binary variable x^k_j to present whether job k is assigned to agent j. Finally, the GAP problem can be formulated as follows.

***Problem** 2:*

$$min \sum_{k \in \mathcal{K}} \sum_{j \in \mathcal{J}} d^k_j x^k_j \tag{11}$$

s.t.

$$\sum_{j \in \mathcal{K}} x^k_j = 1, \forall k \tag{12}$$

$$\sum_{k \in \mathcal{K}} \omega^k_j x^k_j = \Omega^k_j, \forall j \tag{13}$$

$$x^k_j \in [0, 1], \forall j, \forall k \tag{14}$$

Now we show the equivalence of the GAP problem and the simplified version of our proposed problem. If a type n request is generated once on edge node v ($\sum_{v \in \mathcal{V}} m^v_{n,t} = 1$), the communication cost is 0 because we can distribute the request to edge node v, namely where the request origins. Also, we terminate a container immediately after serving a request which means $a^v_{n,t} = 0$. Then, the proposed problem is converted to optimize the switching and running costs. We formulate the simplified problem of cost minimization as follows.

***Problem** 3:*

$$\sum_{v \in \mathcal{V}} \sum_{n \in \mathcal{N}} (p^v_n + \alpha q^v_n) m^v_{n,t} \tag{15}$$

s.t.

$$\sum_{v \in \mathcal{V}} m^v_{n,t} = 1, \forall k, \forall t \tag{16}$$

$$\sum_{n \in \mathcal{N}} u_n m^v_{n,t} \leq U^v_t, \forall v, \forall t \tag{17}$$

$$m^v_{n,t} \in [0, 1], \forall v, \forall n, \forall t \tag{18}$$

If we map job k, agent j, job assignment cost d^k_j, the job size ω^k_j and the resource capacity Ω^k_j in GAP problem to function n, edge node v, system cost $p^v_n + \alpha q^v_n$, function resource demand u_n and resource capacity U^v_t, the equivalence of **Problem** 2 and **Problem** 3 is achieved. Therefore, the GAP problem is a special case of the proposed problem.

Algorithm 1: Request Distribution

1 **foreach** $t \in \mathcal{T}$ **do**
2 **foreach** $v \in \mathcal{V}$ **do**
3 **foreach** $n \in \mathcal{N}$ **do**
4 Sort all nodes v' by $d_{v,v'}$ in ascending order in the sorted list $\mathcal{V}'$;
5 Update the probability of type n function acc. to Equation 22.
6 **if** $\lambda_{v,t}^n \leq a_{v,t}^n$ **then**
7 Assign all $\lambda_{v,t}^n$ requests to current node v.
8 $m_{v,t}^n \leftarrow \lambda_{v,t}^n$.
9 $k_{v,t}^n \leftarrow 0$.
10 **else if** $\lambda_{v,t}^n \geq a_{v,t}^n$ **then**
11 $m_{v,t}^n \leftarrow a_{v,t}^n$.
12 $k_{v,t}^n \leftarrow \lambda_{v,t}^n - a_{v,t}^n$.
13 **foreach** $v' \in \mathcal{V}'$ **do**
14 **if** $d_{v,v'} \leq p_n^v$ **then**
15 **if** $a_{v',t}^n \geq k_{v,t}^n$ **then**
16 $m_{v',t}^n \leftarrow k_{v,t}^n$.
17 $k_{v,t}^n \leftarrow 0$.
18 break;
19 **else**
20 $m_{v',t}^n \leftarrow a_{v',t}^n$.
21 $k_{v,t}^n \leftarrow k_{v,t}^n - a_{v',t}^n$.
22 **end**
23 **end**
24 **end**
25 **if** $k_{v,t}^n \neq 0$ **then**
26 Call Algorithm 2, instantiate $k_{v,t}^n$ new containers at current node v.
27 **end**
28 **end**
29 **end**
30 **end**
31 **end**

As proved above, the proposed request distribution problem is NP-hard. Thus, we propose Algorithm 1 in the next section and prove the theoretical gap of performance.

V. Online Optimization for System Cost

In this section, we first present a request distribution algorithm that assigns serverless requests to suitable edge nodes. The particular reason for this circumstance is that we jointly consider the remaining resource capacity and cached containers on each edge node. Further, we introduce a probabilistic caching algorithm to decide what container to terminate when the resource capacity is close to the limit.

A. Request distribution

Intuitively, a request generated at a particular edge node should be served by cached containers at the current node if the current edge node suffices hardware resources. In this case, the switching cost is reduced to 0 because we can use a cached container to avoid the switching cost. Similarly, the communication cost is also reduced to 0, because the request is assigned to a container on the same edge node. Thus, the proposed algorithm first tries to assign a type n request to the current edge node with a cached type n container. In this paper, we use current edge node to denote the edge node that a request originates from hereafter. If this is not available, the proposed algorithm tries to create a new container of type n at the current edge node or assign the request to a neighbor node with a cached type n container, depending on the switching cost of creating a container and the communication cost to the neighbor node. We provide more details of our algorithm in Algorithm 1 and 2.

Algorithm 2: Probabilistic Caching

1 **while** $\sum_{n\in\mathcal{N}} u_n m_{v,t}^n > U_t^v$ **do**
2 Calculate the probability $P_n(t)$ for each type of container n;
3 Random a container type based on $P_n(t)$;
4 Destroy a container based on the random result;
5 **end**
6 Create a new type n container for the request;

As illustrated in Algorithm 1, when a request arrives at an edge node v, all edge nodes are sorted in ascending order based on the communication cost $d_{v,v'}$ to the current node in line 4. In line 7, if the number of type n requests $\lambda_{v,t}^n$ is smaller than the number of cached containers $a_{v,t}^n$ at the current node v, all the requests are distributed to the current edge node and served by cached containers. In other words, if the current edge node has enough cached containers, we assign all requests to the current node v. Otherwise, in line 16, we distribute the unprocessed requests $k_{v,t}^n$ to neighbor nodes $v' \in \mathcal{V}$ that have sufficient cached containers if the communication cost $d_{v,v'}$ is less than the switching cost p_n^v. The particular reason for this circumstance is that offloading requests to neighbor nodes is more beneficial than creating new containers at the current node in this case. If there still are unprocessed requests after this, in line 26, we create new containers in the current edge node at the cost of extra switching cost by using Algorithm 2. Since the edge node is constrained by resource capacity, cached containers need to be destroyed for new containers when the resource capacity is insufficient in edge nodes. The cached containers are destroyed based on a probabilistic algorithm as shown in Algorithm 2.

Theorem 1: The worst-case system cost for handling a type n request is bounded by $\max\{1 + \frac{p_n^v}{\alpha q_n^v}, 1 + \frac{d_{v,v'}}{\alpha q_n^v}\}$.

Proof 1: The offline optimal solution of distributing a type n request is assigning the request to a cached container in the current edge node. In this case, the switching cost is 0 as a cached container is used. Similarly, the communication cost is also 0 since the request is served at the edge node where it is generated. Thus, the system cost of a type n request is lower bounded by:

$$OPT(n) \geq \alpha q_n^v \tag{19}$$

After that, we consider the worst case produced by Algo-

rithm 1. The worst case includes two cases. The former is that we create a new container at the cost of switching cost p_n^v. The latter is that we offload the request to a nearby edge node at the cost of communication cost $d_{v,v'}$. Hence, we define the maximum system cost of our algorithm as:

$$WORST(n) = \begin{cases} p_n^v + \alpha q_n^v & if \quad d_{v,v'} \geq p_n^v \\ d_{v,v'} + \alpha q_n^v & if \quad d_{v,v'} < p_n^v \end{cases} \tag{20}$$

We define $f(n)$ as $f(n) = WORST(n)/OPT(n)$. Hence, the maximum of $f(n)$ denotes the worst-case competitive ratio. We obtain $f(n)$ for the system cost to serve a request as:

$$f(n) = \begin{cases} 1 + \frac{p_n^v}{\alpha q_n^v} & if \quad d_{v,v'} \geq p_n^v \\ 1 + \frac{d_{v,v'}}{\alpha q_n^v} & if \quad d_{v,v'} < p_n^v \end{cases} \tag{21}$$

Hence, the maximum $f(n)$ achieved by the proposed algorithm is given by $\max\{1 + \frac{p_n^v}{\alpha q_n^v}, 1 + \frac{d_{v,v'}}{\alpha q_n^v}\}$.

B. Probabilistic function caching

In this section, we elaborate on the proposed probabilistic caching policy inspired by a web caching method [44]. The key insight of this paper is that different types of containers should have different probabilities of being evicted from the cache. This is because the resource footprint and invocation frequency vary between different types of containers. When an edge node does not have sufficient hardware resources to create new containers, we use the probabilistic caching policy to decide which container in the cache to destroy. Another benefit of probabilistic caching is to not be overly aggressive with caching the most popular container. By offering a probability to cache less popular containers, we intend to be fair with different kinds of containers.

As illustrated in Algorithm 2, when the required amount of resources exceeds the remaining resource capacity, the proposed algorithm will calculate probabilities $P_n(t)$ for each type of function. Then, we randomly select a container type based on the set of $P_n(t)$. Thus, we destroy a container that is selected. Finally, we repeat this process until there are sufficient resources for creating new containers.

Probability Calculation We propose a probabilistic caching policy, which is contextual, to determine which container to destroy in the cache when an edge node is running close to the capacity. The probability of a type n function being evicted from the cache at time slot t is given by:

$$P_n(t) = \frac{\frac{u_n}{f_n + t_n}}{\sum_{n \in N} \frac{u_n}{f_n + t_n}} \tag{22}$$

Size u_n is the memory footprint of a container as containers are cached in memory. Since the probability of eviction is proportional to the size, large containers are more likely to be evicted than small containers.

Frequency f_n represents the number of times a particular function is invoked until the current time interval. The frequency is updated every time a container is created or destroyed. The probability is inversely proportional to the frequency and hence frequently invoked functions have low probability to be evicted from the cache.

Recency We use t_n to denote the last time a type n function is invoked. t_n reflects the recency a function was invoked. The function, that is recently used, will have a low probability to be evicted from the cache.

Serverless-specific considerations Since the images for all containers of a function type n are identical, it is reasonable to assume that those containers have identical container sizes [30]. Thus, any of the identical containers can be terminated.

C. Time complexity

For each time interval t, the time complexity of pCache is $|\mathcal{V}| \cdot |\mathcal{N}| \cdot (|\mathcal{V}| \cdot \log(|\mathcal{V}| + |\mathcal{V}|)$.

First, Algorithm 1 iterates over $|\mathcal{V}|$ edge nodes and $|\mathcal{N}|$ types of containers which takes $|\mathcal{V}| \cdot |\mathcal{N}|$. Then, Algorithm 1 runs Line 4 to sort all edge nodes in the network which takes $O(|\mathcal{V}| \cdot \log(|\mathcal{V}|))$. The else if part of Algorithm 1 iterate over all the edge nodes in $|\mathcal{V}'|$ which takes $|\mathcal{V}|$ because the size of $\mathcal{V}'$ equals to the size of $\mathcal{V}$. Algorithm 2 may destroy containers based on their probability a few times, making space for caching. Hence, the overall time complexity of pCache is $|\mathcal{V}| \cdot |\mathcal{N}| \cdot (|\mathcal{V}| \cdot \log(|\mathcal{V}| + |\mathcal{V}|)$.

VI. Performance Evaluation

We evaluate the performance of pCache over simulation and implementation in Knative [42], which is an open-source serverless platform that powers enterprise-level solutions. All experiments are repeated 10 times and we report the average of them.

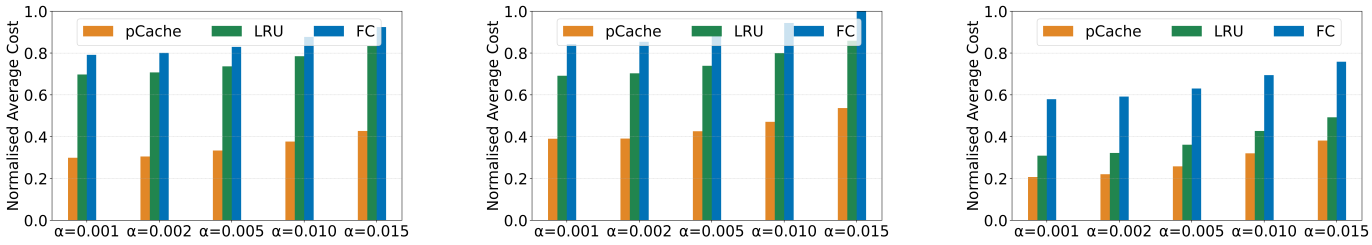
Fig. 3: Edge servers in Melbourne CBD area

A. Simulation setup

To evaluate the performance of the proposed algorithms, we first conducted extensive simulations on a server with 105 GB RAM and an Intel(R) Xeon(R) E5645 processor with 24 cores.

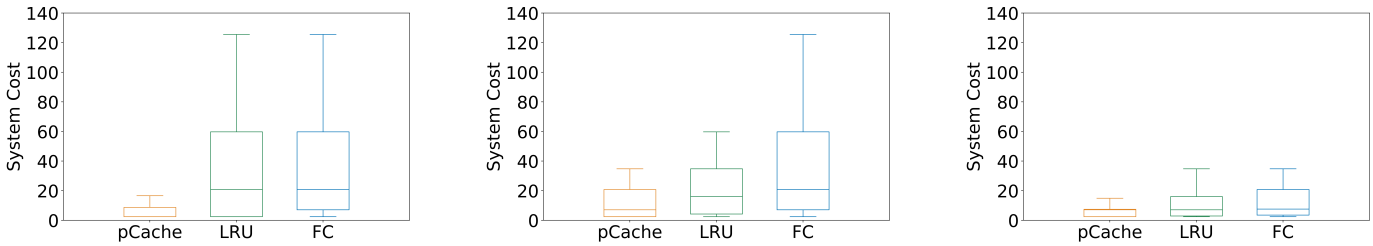
Topology: We use the EUA dataset [45] which includes the information on edge servers in the Melbourne CBD area. The topology consists of 125 edge nodes as illustrated in Figure 3.

Requests: We use the Azure dataset [46] to generate the serverless requests. This dataset contains the invocations of functions on Microsoft Azure for 14 days. The workload at each node is generated using the Zipf distribution [7] with the exponent ranging from 0.5 to 1.5. We use the top four applications in the Azure dataset and map them to four containers shown in Table II.



(a) $\beta = 0.5$ (b) $\beta = 1.0$ (c) $\beta = 1.5$

Fig. 4: Average cost in simulation



(a) $\beta = 0.5$ (b) $\beta = 1.0$ (c) $\beta = 1.5$

Fig. 5: Distributions of system cost in simulation

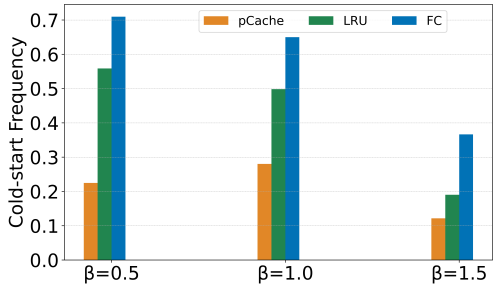


Fig. 6: Cold-start frequency in simulation

Application Name	Memory Size
Web Server	55 MB
File Processing	158 MB
Supermarket Checkout	332 MB
Image Recognition	92 MB

TABLE II: Function instances

Containers: We have built four types of containers obtained from AWS Lambda functions, as shown in Table II. The memory resource allocated to each container is between 55 and 332 MB.

Switching and running cost: According to [47], we set the container switching cost p_n^v inversely proportional to the CPU frequency of node v. Similarly, we set the container running cost q_n^v proportional to the CPU frequency of edge node v. Also, p_n^v and q_n^v are proportional to the container size u_n.

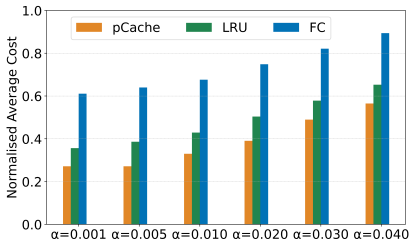
Performance Benchmarks: We select two approaches that are widely used in real-world systems. Least Recently Used (LRU) caching [48] evicts the container that has not been used for the longest time. To implement LRU in our system, each container comes with a time tag to record the last time it has been used. Every time a container has been used, the time tag is refreshed. When the space in the cache is insufficient, pCache will evict the container that is least recently used. Fixed Caching (FC) is widely used in AWS Lambda [49]. It keeps a container alive for a fixed period of time. FC is implemented by assigning each container a Time-To-Live (TTL) tag. Every minute the TTL decreases by 1 until it reaches 0. In the experiments, the caching period was set to 10 minutes, and all the algorithms have the same cache size.

B. Performance in simulation

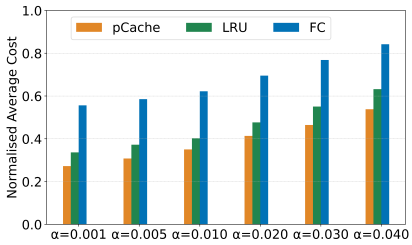
In the simulation, we show the normalized system costs achieved by two benchmarks and our proposed approach in Fig. 4 in which (a), (b) and (c) respectively present the results achieved by setting β as 0.5, 1.0 and 1.5 when considering α as $\{0.001, 0.002, 0.005, 0.010, 0.015\}$.

Performance summary in simulation: pCache reduces the average system cost by up to 57.2% and up to 62.1% compared to LRU and FC, respectively. For cold-start frequency, pCache outperforms LRU and FC by up to 60.8% and 69.1%, respectively. The cold-start frequency is the percentage of requests that experience a cold-start in the experiment. The rationale is that the cached containers of pCache are more frequently reused which can be justified by the cold-start frequency. This actively demonstrates the benefits of capturing context for function caching in serverless computing.

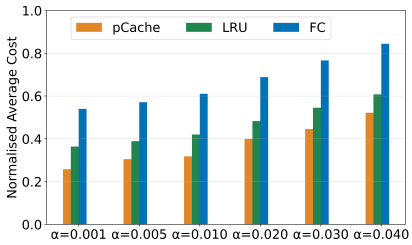
Impact of parameter α: As aforementioned, α is a parameter to tune the trade-off between service latency cost and container running cost. We set α from 0.001 to 0.015. When α is greater than 0.015, the container running cost will be larger than the container switching cost, implying that



(a) $\beta = 0.5$

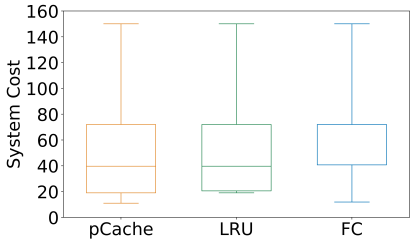


(b) $\beta = 1.0$

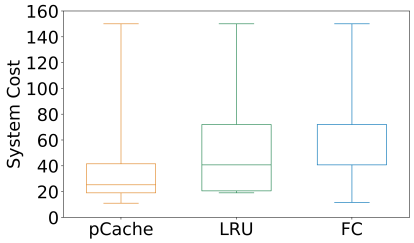


(c) $\beta = 1.5$

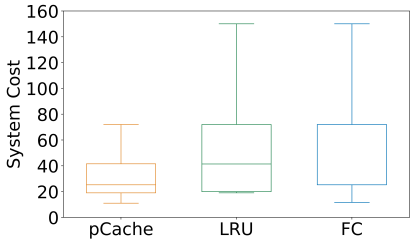
Fig. 7: Average cost in Knative



(a) $\beta = 0.5$



(b) $\beta = 1.0$



(c) $\beta = 1.5$

Fig. 8: Distributions of system cost in Knative

caching functions will always lead to more cost than creating new containers and hence function caching cannot provide any benefits. From Fig. 4, we observe that the parameter α has a noticeable impact on the overall system cost. Overall, when α ranges from 0.001 to 0.015, pCache achieves the best performance where the normalized system cost ranges from 0.3 to 0.43. The system cost is normalized against the maximum system cost achieved in the experiments. In contrast, the system cost of LRU ranges from 0.7 to 0.83. The rationale is that LRU only considers the invocation frequency when evicting functions from the cache. FC achieves 0.79 to 0.92 in normalized system cost because FC cannot dynamically adapt to time-varying workloads.

Impact of parameter β: As aforementioned, the users at each edge node will randomly generate requests conforming to the Zipf-β popularity law [50]. As illustrated in Figure 4, when β ranges from 0.5 to 1.5, pCache always achieves the best performance in system cost. This is because pCache incorporates several characteristics of function invocations such as the container size, the invocation frequency and recency. Compared to LRU and FC, pCache reduces the system cost by up to 57.2% and 62.1%, respectively. The results imply that pCache maintains high performance when coping with dynamic serverless invocations.

Distributions of system cost:

Figure 5a to 5c illustrate the distributions of system cost. The box plots show the maximum, median and minimum of the results when β ranges from 0.5 to 1.5. In figure 5a, the maximum system cost of pCache is 16.6 while that of LRU and FC is 125.6. Similarly, in figure 5b, we observe that pCache achieves 34.8 in the system cost while that of LRU and FC are 59.7 and 125.6, respectively. Figure 5c presents the distributions of system cost for $\beta = 1.5$, pCache reduces the maximum system cost by 57.2% compared to LRU and FC. The rationale is that pCache incorporates the unique features of serverless invocations such as the invocation frequency, recency and etc so that popular containers are more likely to be cached.

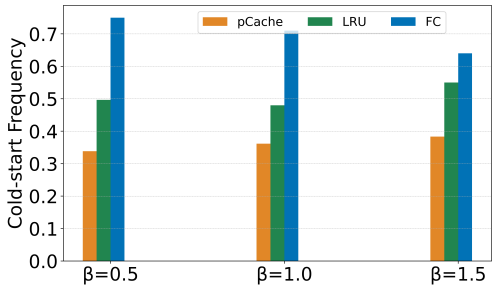


Fig. 9: Cold-start frequency in Knative

Cold-start frequency: Next, we evaluate the impact on cold-start frequency. The cold-start frequency refers to the frequency of cold-start happens across all invocations in the experiment. As shown in Fig. 6, pCache achieves the best performance in terms of cold-start frequency which are 0.224, 0.287, 0.121, respectively. When β ranges from 0.5 to 1.5, LRU achieves 0.558, 0.498 and 0.19, respectively. FC performs the worst due to the rigid caching policy. FC caches each container for a fixed period of time which cannot adapt to burst and concurrent workloads. LRU performs better than FC because LRU considers the recency of invocations which helps to adapt to time-varying request patterns.

C. Implementation in Knative

Knative is an open-source solution for serverless applications over Kubernetes. In Knative, we devise a scheduler to implement the proposed algorithms. Also, we implement the logic of request distribution into the scheduler and dynamically terminate the containers. We deploy Knative over 11 virtual machines with 4GB memory. We reduce the number of requests in the dataset by 10000x to adapt to the system capacity.

Performance summary in Knative: pCache achieves the best performance in terms of average system cost. pCahce reduces the system cost by up to 25% and 55.8% compared to LRU and FC, respectively. Also, we observe that pCache reduces the cold-start frequency by 32% and 51.5% compared to LRU and FC, respectively.

The average system cost: As illustrated in Figure 7, when β increases from 0.5 to 1.5, we observe that pCache always outperforms LRU and FC. The rationale is that pCache assigns high eviction probability to functions that are less likely to be invoked in the near future. Therefore, pCache is more likely to cache popular functions and hence achieves lower cold-start frequency. With lower cold-start frequency, the system cost is effectively reduced. Also, the results imply that pCache shows stable performance over different request distributions when β increases.

In Figure 7, we also observe that when α increases, pCache always shows the best performance. α is a weight parameter that impacts the weight of service latency cost and running cost. When α is very small, e.g., 0.001, we will have the service latency cost significantly dominate the system cost. Hence, in this experiment, we increase the value of α from 0.001 to 0.04 to verify the performance of pCache. The experimental results show that pCache effectively reduces the system cost by up to 55.8% compared to the benchmarks over different values of α.

Distributions of system cost: Figure 8 shows the distributions of system costs in Knative. In Figure 8a, pCache outperforms LRU by 42.4% in light of minimum system cost. Compared to FC, pCache reduces the median system cost by 44.9%. We observe similar trends in Figure 8b and Figure 8c, pCache reduces the minimum system cost by up to 42.4%, the median system cost by up to 64.9%, the maximum system cost by up to 52.1%, respectively. The results in Knative imply that pCache efficiently reduces the system cost in a real-world serverless platform and dynamically adapts to time-varying workloads.

Cold-start frequency: Furthermore, Fig. 9 shows the cold-start frequency obtained by pCache, LRU and FC. pCache still achieves the lowest cold-start frequency followed by LRU and FC. The cold-start frequency of pCache ranges from 0.34 to 0.38 while that of LRU and FC rockets to 0.48 and 0.75, respectively. This is because pCache jointly considers several characteristics of serverless invocations such as the resource footprint, invocation frequency and etc which helps to avoid frequent container creation and termination.

VII. Conclusion and Future Work

In this paper, we studied the online orchestration of serverless functions in edge computing with the aim of minimizing the system cost incurred by request distribution and function caching. We prove this problem is NP-hard by analysis. We propose an online competitive algorithm with performance guarantee which jointly considers network topology and resource constraints. Through extensive evaluations based on trace-driven simulations and implementation over Knative, we show that the proposed algorithms outperform the baselines from state-of-the-art by up to 62.1% in system cost and up to 69.1% in cold-start frequency.

Our work also raises several open questions that are worth investigating in the future. First, we would like to consider the serverless request distribution problem in mobile edge computing and investigate the impact of mobility. Besides the expected challenges incurred by mobile networks, especially with the bandwidth constraints in wireless communication, we intend to further investigate how to extend our framework to more dynamic settings with joint placement and service migration. Second, while we assume that edge nodes never fail, it is worth investigating how to migrate services when an edge node is not available or reliable. In particular, edge servers are not as reliable as those in cloud data centers, making serverless services more vulnerable to failures. Although deploying backup containers seems to be a promising solution to cope with such failures, it dramatically increases the system cost, which violates the spirit of serverless computing, namely pay-as-you-go. Hence, finding the sweet spot between system cost and reliability is pivotal for efficient and reliable serverless platforms. Our proposed framework can be modified to adapt to the extension: we consider keeping functions alive after serving requests, if we further incorporate a failure model, we can then calculate the cost of failures and the cost of provisioning backup containers, aiming to optimize the system cost under reliability constraints.

References

[1] Efterpi Paraskevoulakou and Dimosthenis Kyriazis. Ml-faas: Towards exploiting the serverless paradigm to facilitate machine learning functions as a service. *IEEE Transactions on Network and Service Management*, pages 1–1, 2023.

[2] Xun Shao, Go Hasegawa, Mianxiong Dong, Zhi Liu, Hiroshi Masui, and Yusheng Ji. An online orchestration mechanism for general-purpose edge computing. *IEEE Transactions on Services Computing*, 16(2):927–940, 2023.

[3] Zichuan Xu, Yuexin Fu, Qiufen Xia, and Hao Li. Enabling age-aware big data analytics in serverless edge clouds. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–10, 2023.

[4] Utkalika Satapathy, Rishabh Thakur, Subhrendu Chattopadhyay, and Sandip Chakraborty. Disprotrack: Distributed provenance tracking over serverless applications. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–10, 2023.

[5] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G. Kulkarni, Dan Li, Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 168–181, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386388. doi: 10.1145/3472883.3487014. URL https://doi.org/10.1145/3472883.3487014.

[6] Renchao Xie, Qinqin Tang, Shi Qiao, Han Zhu, F. Richard Yu, and Tao Huang. When serverless computing meets edge computing: Architecture,

challenges, and open issues. *IEEE Wireless Communications*, 28(5): 126–133, 2021. doi: 10.1109/MWC.001.2000466.

[7] Li Pan, Lin Wang, Shutong Chen, and Fangming Liu. Retention-aware container caching for serverless edge computing. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 1069–1078, 2022.

[8] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 780–794, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394208. doi: 10.1145/3544216.3544259. URL https://doi.org/10.1145/3544216.3544259.

[9] Gabriele Russo Russo, Tiziana Mannucci, Valeria Cardellini, and Francesco Lo Presti. Serverledge: Decentralized function-as-a-service for the edge-cloud continuum. In *2023 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 131–140, 2023. doi: 10.1109/PERCOM56429.2023.10099372.

[10] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Mxfaas: Resource sharing in serverless environments for parallelism and efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589069. URL https://doi.org/10.1145/3579371.3589069.

[11] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1489–1504, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5.

[12] Shuiguang Deng, Hailiang Zhao, Zhengzhe Xiang, Cheng Zhang, Rong Jiang, Ying Li, Jianwei Yin, Schahram Dustdar, and Albert Y. Zomaya. Dependent function embedding for distributed serverless edge computing. *IEEE Transactions on Parallel and Distributed Systems*, 33(10): 2346–2357, 2022. doi: 10.1109/TPDS.2021.3137380.

[13] Samuel Kounev, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup, Ian Foster, Prashant Shenoy, Omer Rana, and Andrew A. Chien. Serverless computing: What it is, and what it is not? *Commun. ACM*, 66(9):80–92, aug 2023. ISSN 0001-0782. doi: 10.1145/3587249. URL https://doi.org/10.1145/3587249.

[14] Azure ai. https://azure.microsoft.com/en-gb/solutions/ai/, 2023. 2023.

[15] Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Chengzhong Xu. Serverless computing: State-of-the-art, challenges and opportunities. *IEEE Transactions on Services Computing*, 16(2):1522–1539, 2023. doi: 10.1109/TSC.2022.3166553.

[16] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Specfaas: Accelerating serverless applications with speculative function execution. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 814–827, 2023.

[17] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. Lukewarm serverless functions: Characterization and optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 757–770, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450386104. doi: 10.1145/3470496.3527390.

[18] Lin Gu, Zirui Chen, Honghao Xu, Deze Zeng, Bo Li, and Hai Jin. Layer-aware collaborative microservice deployment toward maximal edge throughput. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 71–79, 2022.

[19] Rong Gu, Xiaofei Chen, Haipeng Dai, Shulin Wang, Zhaokang Wang, Yaofeng Tu, Yihua Huang, and Guihai Chen. Time and cost-efficient cloud data transmission based on serverless computing compression. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–10, 2023.

[20] Xiaojun Shang, Yingling Mao, Yu Liu, Yaodong Huang, Zhenhua Liu, and Yuanyuan Yang. Online container scheduling for data-intensive applications in serverless edge computing. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–10, 2023.

[21] Claudio Cicconetti, Marco Conti, and Andrea Passarella. A decentralized framework for serverless edge computing in the internet of things. *IEEE Transactions on Network and Service Management*, 18(2):2166–2180, 2021. doi: 10.1109/TNSM.2020.3023305.

[22] Yuepeng Li, Deze Zeng, Lin Gu, Mingwei Ou, and Quan Chen. On efficient zygote container planning toward fast function startup in serverless edge cloud. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–9, 2023.

[23] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Trans. Softw. Eng. Methodol.*, 32(5), jul 2023. ISSN 1049-331X.

[24] Lin Gu, Deze Zeng, Jie Hu, Bo Li, and Hai Jin. Layer aware microservice placement and request scheduling at the edge. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–9, 2021. doi: 10.1109/INFOCOM42981.2021.9488779.

[25] Home - knative. https://knative.dev/docs/, 2023.

[26] Xuanyu Cao, Junshan Zhang, and H. Vincent Poor. An optimal auction mechanism for mobile edge caching. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 388–399, 2018.

[27] C. Chen, L. Nagel, L. Cui, and F. Tso. S-cache: Function caching for serverless edge computing. In *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '23, page 1–6, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700828. doi: 10.1145/3578354.3592865.

[28] Mohammad Sadegh Aslanpour, Adel N. Toosi, Muhammad Aamir Cheema, and Raj Gaire. Energy-aware resource scheduling for serverless edge computing. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 190–199, 2022. doi: 10.1109/CCGrid54584.2022.00028.

[29] J. Shen, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu. Defuse: A dependency-guided function scheduler to mitigate cold starts on faas platforms. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021.

[30] Alexander Fuerst and Prateek Sharma. Faascache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 386–400, New York, NY, USA, 2021.

[31] Konstantinos Poularakis, Jaime Llorca, Antonia M. Tulino, Ian Taylor, and Leandros Tassiulas. Joint service placement and request routing in multi-cell mobile edge computing networks. In *IEEE INFOCOM 2019*, pages 10–18, 2019.

[32] Changyuan Lin and Hamzeh Khazaei. Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):615–632, 2021.

[33] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259–271, 2021. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2020.07.017. URL https://www.sciencedirect.com/science/article/pii/S0167739X2030399X.

[34] Changyuan Lin, Nima Mahmoudi, Caixiang Fan, and Hamzeh Khazaei. Fine-grained performance and cost modeling and optimization for faas applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):180–194, 2023. doi: 10.1109/TPDS.2022.3214783.

[35] John Tadrous and Atilla Eryilmaz. On optimal proactive caching for mobile networks with demand uncertainties. *IEEE/ACM Transactions on Networking*, 24(5):2715–2727, 2016. doi: 10.1109/TNET.2015.2478476.

[36] Junzhong Jia, Lei Yang, and Jiannong Cao. Reliability-aware dynamic service chain scheduling in 5g networks based on reinforcement learning. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021. doi: 10.1109/INFOCOM42981.2021.9488707.

[37] V. Farhadi, F. Mehmeti, T. He, T. F. La Porta, H. Khamfroush, S. Wang, K. S. Chan, and K. Poularakis. Service placement and request scheduling for data-intensive applications in edge clouds. *IEEE/ACM Transactions on Networking*, 29(2):779–792, 2021.

[38] Stephen Pasteris, Shiqiang Wang, Mark Herbster, and Ting He. Service placement with provable guarantees in heterogeneous edge computing systems. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 514–522, 2019.

[39] Ruiting Zhou, Xiaoyi Wu, Haisheng Tan, and Renli Zhang. Two time-scale joint service caching and task offloading for uav-assisted mobile edge computing. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 1189–1198, 2022. doi: 10.1109/INFOCOM48880.2022.9796714.

[40] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 753–767, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051.

[41] Kubernetes. https://kubernetes.io/, 2023.

[42] Kourier. https://github.com/knative-extensions/net-kourier, 2023.

[43] P.C. Chu and J.E. Beasley. A genetic algorithm for the generalised assignment problem. *Computers & Operations Research*, 24(1):17–23, 1997. ISSN 0305-0548.
[44] Ludmila Cherkasova. Improving www proxies performance with greedy-dual-size-frequency caching policy. In *HP Labs Technical Report 98-69*, 1998.
[45] Phu Lai, Qiang He, Mohamed Abdelrazek, Feifei Chen, John Hosking, John Grundy, and Yun Yang. Optimal edge user allocation in edge computing with variable sized vector bin packing. In Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu, editors, *Service-Oriented Computing*, pages 230–245, Cham, 2018. Springer International Publishing. ISBN 978-3-030-03596-9.
[46] Azure. Azurepublicdataset. https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md, 2019.
[47] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853.
[48] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. Ofc: An opportunistic caching system for faas platforms. EuroSys '21, page 228–244, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383349. doi: 10.1145/3447786.3456239. URL https://doi.org/10.1145/3447786.3456239.
[49] Aws lambda. https://aws.amazon.com/lambda/, 2023.
[50] Samta Shukla, Onkar Bhardwaj, Alhussein A. Abouzeid, Theodoros Salonidis, and Ting He. Proactive retention-aware caching with multi-path routing for wireless edge networks. *IEEE Journal on Selected Areas in Communications*, 36(6):1286–1299, 2018.

Chen Chen is a Research Associate in the System Research Group, Department of Computer Science and Technology, University of Cambridge. His research interests are broadly in areas of serverless computing, cloud and edge computing, network resource orchestration, in-network computing, distributed systems and service chaining.

Manuel Herrera is a Senior Research Associate in Distributed Intelligent Systems at the University of Cambridge. His research focuses on the development of predictive analytics and complexity science for smart and resilient critical infrastructure such as transport, telecommunications, and water. He is a fellow of the Royal Statistical Society (RSS), a committee member of the RSS Special Interest Group in statistical engineering, and the recipient of the Frank Hansford-Miller fellowship 2021 in applied statistics, awarded by the Statistical Society of Australia (WA Branch).

Ge Zheng is a Research Associate in the Distributed Information and Automation Laboratory (DIAL) at the Institute for Manufacturing (IfM), Department of Engineering, University of Cambridge, UK. She received her PhD degree with full PhD scholarship in the Department of Computing and Informatics at the University of Bournemouth, UK, and MSc degree with Academic Excellence International Masters Scholarship in Electronic Engineering from the School of Computer Science and Electronic Engineering at the University of Essex. Her interest areas involve edge computing, supply chain risk prediction, pattern recognition and/or classification, intelligent transportation systems, and healthcare applications.

Liqiao Xia received the B.S. degree from South China Agricultural University in 2017, M.S. degree from Hong Kong University of Science and Technology in 2020. He is currently working towards a Ph.D. degree in the Department of Industrial and Systems Engineering at the Hong Kong Polytechnic University. His research interests include edge computing, knowledge graph and reliability analysis.

Zhengyang Ling is a Research Associate at the Distributed Information and Automation Laboratory in University of Cambridge. He earned his PhD from the University of Leeds and holds MSc and bachelor degrees from East China University of Science and Technology. With over three years of experience in the automotive industry, including roles at SAIC Motor and ChanganUK. His research interests are edge computing, wireless sensors and actuators, optical measurement and their applications.

Jiangtao Wang received the PhD degree from Peking University, China, in 2015. He is currently an Associate Professor in the Intelligent Healthcare Centre, Coventry University, UK. Before that, he was a lecturer with the School of Computing and Communications at Lancaster University, UK. His research interest includes mobile and pervasive computing, crowdsensing/crowdsourcing, and digital health.