Journal of Network and Computer Applications 251 (2026) 104490

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Research paper

FaasOrc: A bi-level function scheduling and caching framework for serverless edge computing

Chen Chen [a], Lars Nagel [b], Lin Cui [c], Weijia Jia [d], Fung Po Tso [b,*]

[a] Department of Computer Science and Technology, University of Cambridge, Cambridge, UK
[b] Department of Computer Science, Loughborough University, Loughborough, UK
[c] Department of Computer Science, Jinan University, Guangzhou, China
[d] Beijing Normal University (BNU Zhuhai) and BNU-HKBU United International College, Zhuhai, China

ARTICLE INFO

Keywords:
Serverless computing
Edge computing
Coldstart
Container caching

ABSTRACT

Serverless computing, underpinned by an event-driven approach with transient stateless containers, significantly enhances resource efficiency and simplifies function development. To maintain an acceptable Quality of Service (QoS) agreed in the Service Level Agreement (SLA), service providers need to improve the response latency while considering resource efficiency. However, cold-start delays in container initialization often lead to considerable latency in these applications. Existing mitigation strategies, such as pre-warming and function caching, are inadequate due to workload skewness and oscillation across edge nodes. These limitations are particularly critical in resource-constrained edge environments. We must jointly consider multiple factors, such as node status, function resource requirement and function popularity.

To overcome these limitations, this paper presents *FaasOrc*, a bi-level function orchestration framework to mitigate workload skewness and oscillation across edge nodes. *FaasOrc* uses a cluster-level scheduler to schedule requests and a node-level manager to detect popular functions. Our comprehensive evaluation, consisting of two parts: simulations and a real-system prototype over ***Knative***, benchmarks the proposed solution against existing scheduling and caching strategies. The findings highlight our method's capability to reduce the response latency by 31.4%.

1. Introduction

Edge computing employs distributed edge nodes with limited hardware resources and allows the computation of tasks close to where they are generated (e.g., sensing/actuation). Serverless functions are usually single-purpose and stateless containers, created only when responding to event-driven tasks (Lin et al., 2024; Suyi Li et al., 2023; Moakhar and Abrishami, 2024; Emami Khansari and Sharifian, 2025; Shujairi, 2025). As a result, the users benefit from the low latency and flexible cost model in the serverless paradigm (Sakirin and Asif, 2023) which is impacted by the cold-start issue. Given its high potential for efficient resource management and effortless scalability, the serverless paradigm is promising for accelerating the growth of edge computing (Tütüncüoğlu et al., 2024; Ziyi Wang et al., 2023; Yue et al., 2024; Yuepeng Li et al., 2023). Serverless services are provided by all major computing platforms such as Amazon (Lambda, 2023), Google Functions (Google Cloud, 2023a) and Microsoft Azure (Azure Functions, 2023) which offer API services, web services, image recognition, crowd density measurement and many others (Yu et al., 2023; Xu et al., 2023; Lee et al., 2025; Yang et al., 2025).

A key challenge in serverless edge computing is the cold-start delay incurred by the initialization of the container (Roy et al., 2022; Pan et al., 2022; Karamzadeh and Shameli-Sendi, 2024; Choi and Park, 2022; Hasan et al., 2025). This startup process takes a considerable amount of time which adds to the overall latency observed by the user. The cold-start issue impairs a main advantage of edge computing, namely the reduction of latency and thus real-time responsiveness (Qi et al., 2022; Shen et al., 2021). In the case of interactive services, the end-to-end latency usually must meet a Service-Level Object (SLO) target in the order of a few tens of milliseconds (Google Cloud, 2023b).

Existing solutions for tackling cold-start include layer sharing (Gu et al., 2021; Akkus et al., 2018), function pre-warming and function caching (Roy et al., 2022; Pan et al., 2022). **Layer sharing:** Some work (Cheng et al., 2024; Cadden et al., 2020) resort to layer sharing to shorten the latency of the cold-start process. However, layer sharing

* Corresponding author.
E-mail addresses: cc2181@cam.ac.uk (C. Chen), l.nagel@lboro.ac.uk (L. Nagel), tcuilin@jnu.edu.cn (L. Cui), jiawj@uic.edu.cn (W. Jia), p.tso@lboro.ac.uk (F.P. Tso).

https://doi.org/10.1016/j.jnca.2026.104490
Received 8 August 2025; Received in revised form 21 January 2026; Accepted 14 April 2026
Available online 25 April 2026

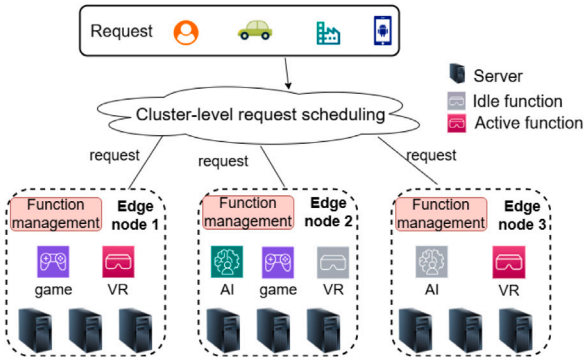


Fig. 1. An example of function scheduling in serverless edge computing, illustrating request scheduling and function lifetime management.

reduces the initialization time but only partially mitigates cold-starts as it decreases startup time without preventing cold-start occurrences. **Function pre-warming:** Function pre-warming refers to creating a new container before a request arrives and hence avoiding the occurrence of cold starts. Several works (Mahgoub et al., 2022; Zhou et al., 2022; Yang et al., 2022) propose different machine learning models to predict the arrival patterns of serverless invocations. These methods are highly dependent on prediction accuracy, necessitating system knowledge and substantial modeling effort. **Function caching:** Large cloud providers (e.g., AWS and Azure) adopt a fixed caching policy that retains serverless functions for 10 or 20 min after executing a job (Shahrad et al., 2020). Consequently, future invocations can be served by cached containers and hence cold-starts are avoided. Many research efforts (Yang et al., 2024; Yu et al., 2024; Fuerst and Sharma, 2021) have proposed different techniques to detect hotspot functions and manage the functions' lifetime. However, function caching needs to be carefully designed to avoid over-provisioning, which wastes scarce system resources. Although Deep Reinforcement Learning (Agarwal et al., 2024; Alqaraghuli and Karan, 2024) is another promising approach, we selected lightweight heuristics to achieve real-world deployability.

However, existing works mainly focus on function management at the node level and ignore the interplay between request scheduling and function management. Request scheduling refers to assigning a request to a specific function at an edge node (also known as load balancing), while function management refers to the lifetime management of functions on each edge node. Ideally, traffic needs to be distributed to the nearest edge node, reducing the distance data travels and the latency. As illustrated in Fig. 1, the cluster-level scheduling dynamically alters the workload on each edge node. Consequently, local function management decisions are closely interconnected with cluster-level request scheduling decisions. The decision-making process must consider both the cluster-level request scheduling and node-level container management.

At first glance, these challenges might seem addressable by load-balancing future requests to resource-redundant edge nodes or caching popular functions locally. The former fails to reuse idle functions for the future while the latter has only a local view of functions, both leading to suboptimal decisions.

Existing research overlooks the complex coupling relationship between request scheduling and function management decisions while only focusing on the single decision optimization. Thus, optimizing response latency by a single decision cannot achieve the effect of joint decision optimization. For example, workload skewness (Yang et al., 2024) becomes a common phenomenon across different edge nodes because edge computing leans toward deploying services close to the users. Hence, function management at the node level is massively correlated to request scheduling at the cluster level. Coordinating them requires considering the node workload, function resource requirement and popularity. Moreover, the request patterns can oscillate over time and incur resource contention between functions, further complicating the cluster management.

In this work, we capture the different levels of the system behavior and architecturally correlate the cluster-level request scheduling and node-level function management. We design *FaasOrc*, a bi-level orchestration framework for latency-sensitive services in serverless edge computing. The goal is to better use global information to help each node handle workload skewness and oscillation. *FaasOrc* achieves this by (1) load-balancing based on communication delay and node workload. (2) popular function detection at the node level through global information such as the invocation counts of a function over a period of time. In this way, request scheduling and function lifetime management complement each other by exchanging information periodically (Lafta and Al-fiskhaltom, 2023).

In the cluster, *FaasOrc* runs a centralized scheduler called *Commander*. Commander observes the incoming requests and dispatches them to different edge nodes based on the communication delay and node status, using a lightweight online heuristic algorithm with performance guarantee (Section 5.1). Commander performs fast and fine-grained request scheduling using global information and the assistance from local function managers. At each edge node, *FaasOrc* locally runs a lightweight function manager called *Beacon*. Beacon fetches global invocation information from Commander, uploads node status to Commander and manages the lifetime of local functions based on a number of invocation characteristics (Section 5.2).

Our main contributions are summarized as follows.

- We identify the correlation between request scheduling and function management, highlighting that achieving their complementarity is non-trivial.
- We devise *FaasOrc* to co-optimize the cluster-level request scheduling and node-level function caching in a unified architecture for serverless edge computing.
- *FaasOrc* is a bi-level orchestration framework designed to account for distinct levels of system behavior in serverless edge computing. It contributes to the state of the art by *(i)* separately designing mechanisms for cluster-level request scheduling and node-level function management, while incorporating a popularity-based policy to identify frequently used functions; and *(ii)* combining global and local information to effectively handle workload skewness and oscillations in the cluster.
- We implement *FaasOrc* in an open-source FaaS platform ***Knative*** (Knative, 2023) as well as in ns-3 (nsnam, 2025) and evaluate *FaasOrc* using production FaaS traces from Microsoft Azure Functions (Azure, 2023). Results show that *FaasOrc* outperforms the state-of-the-art frameworks by 31.4% in terms of response latency.

The remainder of this paper is organized as follows. We first provide an overview of the related work (Section 2) for serverless edge computing, including request scheduling, function pre-warming and caching. We summarized the existing solutions and identified the limitations of the state-of-the-art. After that, we present the system model (Section 3) and the problem (Section 4). Subsequently, we present the proposed framework (Section 5), consisting of request scheduling and function management. Finally, we justify the superiority of the proposed framework in the experiments (Section 6) and draw the conclusion (Section 7).

2. Related work

In recent years, serverless computing has attracted significant attention. In particular, much research focuses on request scheduling, function pre-warming and function caching.

2.1. Request scheduling in serverless edge computing

A number of studies have investigated request scheduling in edge computing. Shang et al. (2023) investigate the function placement and data routing problem, fully incorporating the heterogeneity of edge networks and the overheads of serverless computing. Poularakis et al. (2019) propose a service placement and flow routing algorithm based on randomized rounding to maximize the number of requests handled by base stations in mobile edge computing. Farhadi et al. (2021) propose a two-time-scale algorithm to jointly optimize the service placement and request scheduling. Gu et al. (2022) present a layer-aware container placement and request scheduling algorithm, aiming to maximize the number of placed containers and the overall throughput. Cheng et al. (2024) use layer-wise memory sharing and directed acyclic graphs to reduce cluster memory footprint for serverless workflows. Pan et al. (2022) map the function caching problem to the ski-rental problem to optimize the deployment cost. SMore (Liu et al., 2025) optimizes GPU resource usage for Deep Learning clusters by co-locating functions. Many other works (He et al., 2018; Pasteris et al., 2019; Gao et al., 2019) have also extensively examined the service placement problem in edge computing. Flame (Yang et al., 2024) uses round-robin as the load-balancing policy, but it falls short of understanding workload skewness and communication delays between edge nodes. Flame also uses a centralized cache system with an exponentially decaying algorithm to learn invocation characteristics. Agarwal et al. (2024) combines long-short term memory with proximal policy optimization to capture environment parameters and efficiently scale serverless functions. Joosen et al. (2025) introduce a pod utility ratio to measure the pod's execution time relative to its cold start. AFaaS (Chai et al., 2025) identify three sources of overlooked latency, including control path latency, resource contention latency and initialization latency. AFaaS uses resource pooling and lightweight runtime to accelerate the cold-start process. Golgi (Suyi Li et al., 2023) uses nine low-level metrics for runtime performance to predict function performance and thus optimizes the function scheduling in cloud computing. QUART (Lin et al., 2024) optimizes the key stages in pipeline parallelism and caches parameters for large language models. Demeter (Yue et al., 2024) uses multi-agent reinforcement learning to co-optimize function placement and resource allocation in serverless edge computing. Rangarajan and Al-Quraishi (2023) highlights the need for multi-layered orchestration in IoT system.

However, these works largely overlook the interplay between request scheduling and function management, making them insufficient to optimize response latency in serverless computing. For instance, requests can be assigned to an edge node with idle functions to avoid the cold start process.

2.2. Function pre-warming

Function pre-warming is to create a new container before a request arrives and hence avoid the occurrence of cold starts. This method usually uses a prediction model to decide when to pre-warm a function. Thus, the performance of "pre-warming" is largely decided by the prediction accuracy. ORION (Mahgoub et al., 2022) employs a latency model to estimate the request arrivals. Similarly, AQUATOPE (Zhou et al., 2022) and INFless (Yang et al., 2022) resort to Bayesian Neural Network and Long-Short Term Histogram to predict the arrival pattern, respectively. Shahrad et al. (2020) use an Arima model (Pmdarima, 2025) to predict the invocation pattern and combine a TTL (time-to-live) based caching policy to reduce the occurrence of cold starts. Chen et al. (2025) use Q-learning and Proximal Policy Optimization to schedule serverless containers in edge computing.

However, prediction models face significant challenges under spiky workloads. Thanks to the lightweight popularity score, *FaasOrc* adaptively increases and decreases the number of cached containers and therefore reduces the wastage of resource allocation.

2.3. Function caching

FaasCache (Fuerst and Sharma, 2021) uses a priority-based greedy caching policy, also considering the characteristics of functions. However, key differences exist: (1) FaasCache only considers a single-server setting without considering request scheduling algorithms (We proposed *Commander* in Section 5). (2) FaasCache has no termination policy if a container with high priority is idle for a period of time, falls short of efficient resource utilization. (3) A cumulative frequency is used which exhibits limitations for time varying workloads (Function popularity can vary over time). RainbowCake (Yu et al., 2024) proposes to use a layer-wise container pre-warming and caching approach, aiming to optimize cold-starts. RainbowCake classifies the startup process of containers into three stages and manages different stages individually. Roy et al. (2022) use low-cost nodes to cache containers which keeps a larger number of containers warm under the same cost budget. Function caching requires extra memory and hence it must to be carefully designed to avoid wasting resources.

In contrast to these existing solutions, *FaasOrc* uses a bi-level structure which co-optimizes the cluster-level request scheduling and the node-level function management. We use lightweight heuristics to control the online decisions and include four important characteristics of serverless computing, i.e., the recency, invocation count, cold-start delay and memory allocation. *FaasOrc* focuses on workload skewness and hotspot functions, trade-off between communication delays and cold-start delays.

2.4. Edge cloud continuum

Prior work on serverless edge computing has explored platforms and scheduling mechanisms that extend Function-as-a-Service (FaaS) from centralized clouds to geographically distributed edge nodes. In multi-operator environments, where edge resources are owned by independent administrative domains, individual compute clusters incur private expenses for CPU cycles, memory reserved for warm instances, storage for function images and data, and energy consumption. Existing efforts (Filinis et al., 2024; Pandey et al., 2025; Russo Russo et al., 2024) typically assume cooperative operators or centralized control, overlooking the strategic behavior of rational nodes. As a result, without explicit incentive mechanisms, operators may free-ride by avoiding costly actions such as pre-warming functions, caching low-popularity images, or admitting latency-sensitive workloads, thereby degrading system-wide responsiveness. To solve the problem in the multi-operator environment, auction-based approaches (Fei Wang et al., 2023; Zhang et al., 2025) are efficient mechanisms to coordinate multiple stakeholders in the edge environment. Another work (Zhang and Jacobsen, 2025) uses a distributed hash table to record geographical and operator context for local resource scheduling.

FaasOrc is orthogonal to these frameworks and can be integrated with these frameworks to support *FaasOrc* function caching policy in each administrative domain.

3. System model

We first present the system model used for problem formulation. Then, we describe the system design of the proposed framework.

3.1. Overview of system model

We consider an edge network consisting of multiple edge nodes at different geographical locations which we refer to as a cluster. These edge nodes have computational resources and serve service requests. Requests are generated by end users, and requests are processed by functions hosted by edge nodes. The serverless system forwards requests to edge nodes of the cluster based on its placement policy.

Table 1
Symbols and variables.

Symbols	Description
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Physical network graph
$\mathcal{V}$	Edge nodes in the network
$\mathcal{E}$	Physical links in the network
C_v	Memory capacity at node $v \in \mathcal{V}$
C_v^{rem}	Remaining memory capacity at node $v \in \mathcal{V}$
c_n	Amount of memory required by function of type $n \in \mathcal{N}$
$\mathcal{N}$	Set of serverless functions
$a_{v,t}^n$	Number of type n cached containers at node v in the beginning of time interval t
$b_{v,t}^n$	Number of active type n containers at node v in time interval t
$l_{v,v'}$	Communication delay between node v and v'
p^n	The processing delay of function n
d^n	The cold-start delay of function n
$\lambda_{v,t}^n$	Number of type n requests generated at node v in time interval t
$k_{v,t}^n$	Number of type n requests remaining to be deployed at time interval t
α	Parameter to reduce the number of invocations in dataset
β	The Zipf skewness parameter
l_n	The time of n has not been invoked.
Decision variables	
$m_{v,t}^n$	Number of type n requests distributed to node v at t
$m_{v\to v',t}^n$	Number of type n requests distributed from v to v' at t
$y_{v,t}^n$	Number of type n containers to be terminated after time interval t

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph representing the physical network which consists of a set $\mathcal{V}$ of edge nodes and a set $\mathcal{E}$ of links. Each edge node $v \in \mathcal{V}$ is equipped with an amount of hardware resources (e.g., CPU and memory) denoted by C_v. $\mathcal{R}$ denotes the set of requests. Edge nodes in the system process incoming requests by invoking serverless containers. The set of container types is denoted by $\mathcal{N}$. The total execution time T is divided into time intervals $t \in \mathcal{T} = \{1, \dots, T\}$. The lifecycle of each container can span multiple time intervals.

In the system, containers that are alive but not executing tasks are referred to as *cached containers* hereafter. Similarly, containers that are alive and executing tasks are referred to as *active containers*.

4. Problem description

We consider three types of delays that are essential to the system performance: communication delay, processing delay and cold-start delay. All symbols and variables are listed in Table 1. If a request is generated at edge node v and is distributed to edge node v', we use $\ell_{v,v'}$ to denote the expected communication delay between v and v' which is proportional to the geographical distance between v and v'. Let $m_{v\to v',t}^n$ denote the number of type n requests generated at node v and distributed to node v' in time interval t. Then the total communication delay D_t^L in time interval t is:

$$D_t^L = \sum_{v,v' \in \mathcal{V}} \sum_{n \in \mathcal{N}} \ell_{v,v'} m_{v\to v',t}^n. \tag{1}$$

The processing delay of a function is incurred by its processing time at a type n container which is proportional to the amount of traffic in a request. Let p^n denote the expected processing time of a request with the type n function. Let $m_{v,t}^n$ denote the number of type n requests distributed to node v in time interval t. Then the total processing delay D_t^P in time interval t is:

$$D_t^P = \sum_{v \in \mathcal{V}} \sum_{n \in \mathcal{N}} p^n m_{v,t}^n. \tag{2}$$

The cold-start delay is incurred by the container initialization. By d^n we denote the expected cold-start delay for instantiating a single container of type n. Let $y_{v,t-1}^n$ denote the number of containers that were destroyed on edge node v at the end of time interval $t-1$ and $a_{v,t}^n$ the number of cached type n containers on edge node v at the beginning of time interval t. Then $a_{v,t}^n = \max\{a_{v,t-1}^n, m_{v,t-1}^n\} - y_{v,t-1}^n$. The total cold-start latency D_t^C of all requests in time interval t is:

$$D_t^C = \sum_{v \in \mathcal{V}} \sum_{n \in \mathcal{N}} d^n max\{m_{v,t}^n - a_{v,t}^n, 0\}. \tag{3}$$

The total latency D_t of the system is the sum of all delays:

$$D_t = 2D_t^L + D_t^P + D_t^C. \tag{4}$$

4.1. Problem formulation

The objective is to minimize the end-to-end latency taken over all time intervals. This problem can be formulated as an integer linear programming problem:

Problem 1.

$$min \quad \sum_{t=1}^{T} D_t, \tag{5}$$

$$s.t. \quad \sum_{n \in \mathcal{N}} c_n (max\{a_{v,t}^n, m_{v,t}^n\} + b_{v,t}^n) \le C_v, \quad \forall v, \forall t, \tag{6}$$

$$y_{v,t}^n \in [0, max\{a_{v,t}^n, m_{v,t}^n\}], \quad \forall v, \forall n, \forall t, \tag{7}$$

$$\sum_{v' \in \mathcal{V}} m_{v\to v',t}^n = \lambda_{v,t}^n, \quad \forall v, \forall n, \forall t, \tag{8}$$

$$m_{v,t}^n, m_{v\to v',t}^n \in \{0, 1, 2, \dots, \lambda_{v,t}^n\}. \tag{9}$$

The first constraint ensures that the hardware resources requested from v do not exceed the resource limit C_v (where c_n denotes the amount of resource required by a type n container). The hardware resources can be heterogeneous and therefore increase the complexity of the problem. We also considered this in our experiment as well to show the feasibility of our algorithms in real-world prototype. $a_{v,t}^n$ represents the number of cached containers at node v. $b_{v,t}^n$ denotes the number of active containers that are executing tasks in time slot t. The second constraint guarantees that the number of destroyed containers is not greater than the number of cached containers. The third constraint ensures that each request is distributed to only one edge node; $\lambda_{v,t}^n$ is the number of type n requests generated from edge node v in time interval t. The last constraint ensures the decision variables are integer.

5. The *FaasOrc* framework

In this section, we propose *FaasOrc*, a practical request scheduling and function management framework for minimizing response latency in serverless computing. *FaasOrc* leverages lightweight heuristics to make decisions and collect information efficiently and thereby provide good scalability as shown in the experiment section.

FaasOrc consists of two key components: Commander and Beacon as illustrated in Fig. 2. *Commander* is a cluster-level scheduler, and *Beacon* is a per-node function manager.

Fig. 3 shows the overall workflow of *FaasOrc*. *FaasOrc* first decides which edge node to host a request. After that we decide whether to reuse a cached container or create a new one. Finally, in the end of each time interval, we decide the termination of containers. Meanwhile, Beacon periodically exchanges information with Commander.

5.1. FaasOrc Commander

At the cluster level, we run a Commander instance, which periodically receives node status from the Beacon to make decisions. Commander implements the following three functionalities. (1) It communicates with Beacon, exchanging the information of node workload, function status, and total invocation count of each function. (2) It schedules requests to different edge nodes when requests arrive. (3)

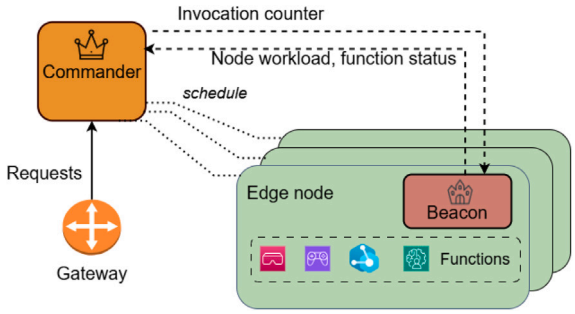


Fig. 2. *FaasOrc* uses a bi-level scheduling framework. The cluster-level controller (Commander), monitoring workloads and schedule requests to edge nodes; the node-level manager Beacon, observing node status and managing container lifetime. Commander and Beacon exchanges information periodically such as node workload, function status, and invocation counter.

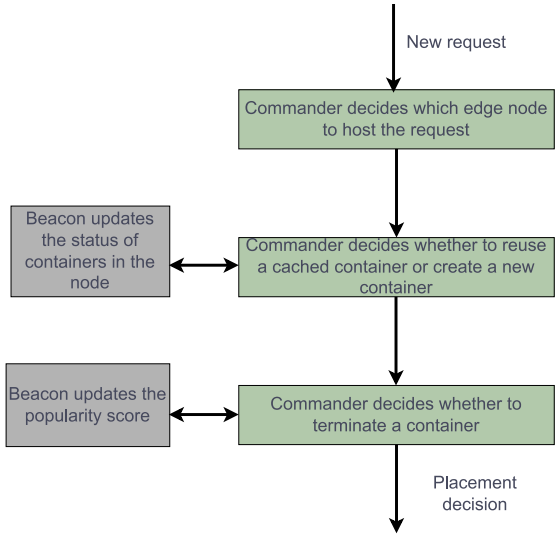


Fig. 3. The workflow of *FaasOrc*.

Algorithm 1: Request schedule()

Input : $\lambda^n_{v,t}$
Output: $m^n_{v,t}$, $m^n_{v\to v',t}$

1 **foreach** $n \in \mathcal{N}$ **do**
2 Sort all nodes v' by $\ell_{v,v'}$ in ascending order and store them in the sorted list $\mathcal{V}'$;
3 Update the popularity score $H^n_v(t)$ for type n containers acc. to Eq. (10);
4 **if** $a^n_{v,t} - \lambda^n_{v,t} \geq 0$ **then**
5 Assign $\lambda^n_{v,t}$ requests to cached containers, $m^n_{v,t} \leftarrow \lambda^n_{v,t}$;
6 $k^n_{v,t} \leftarrow 0$;
7 **else**
8 Assign part of requests to cached containers, $m^n_{v,t} \leftarrow a^n_{v,t}$;
9 $k^n_{v,t} \leftarrow \lambda^n_{v,t} - a^n_{v,t}$;
10 **foreach** $v' \in \mathcal{V}'$ **do**
11 **if** $2\ell_{v,v',t} \leq d^n$ **then**
12 **if** $a^n_{v',t} \geq k^n_{v,t}$ **then**
13 Distribute all remaining requests to neighbor node v';
14 $m^n_{v\to v',t} \leftarrow k^n_{v,t}$;
15 $k^n_{v,t} \leftarrow 0$;
16 **break**;
17 **else**
18 Distribute $a^n_{v',t}$ remaining requests to neighbor node v';
19 $m^n_{v\to v',t} \leftarrow a^n_{v',t}$;
20 $k^n_{v,t} \leftarrow k^n_{v,t} - a^n_{v',t}$;
21 **end**
22 **end**
23 **end**
24 **if** $k^n_{v,t} \neq 0$ **then**
25 Call Algorithm 2 to instantiate $k^n_{v,t}$ new containers at current node v;
26 **end**
27 Call Beacon to update the popularity for type n function acc. to Eq. (10);
28 **end**
29 **end**

It decides which function will serve a specific request. By combining global and local information, Commander can efficiently handle workload skewness and reuse cached functions.

We now introduce the pseudocode of Commander as illustrated in Algorithm 1, Commander first sorts the edge nodes in a list $\mathcal{V}'$ based on the communication delay to the current node and calculates the popularity score for each type of container (Lines 2–3). This step is to make sure that requests are executed in functions close to them, avoiding long communication delay.

If $a^n_{v,t} - \lambda^n_{v,t} \geq 0$, which means cached containers suffice to serve all requests, the algorithm assigns all incoming requests $\lambda^n_{v,t}$ to cached containers. Correspondingly, the number of left requests $k^n_{v,t}$ is set to 0 (Line 4–6). The else part indicates that cached containers in edge node v are not sufficient to accommodate all requests; the philosophy is to use cached containers as possible and assign the rest to neighbor nodes by iterating $\mathcal{V}'$ (Line 8). This will help to decide which edge node to use according to the communication delay between the current edge node and its neighbor nodes.

Because the current edge node v lacks sufficient cached containers, we assign the remaining requests, that cannot be processed by caching containers on the current edge node v, to neighbor nodes where the communication delay between the current node and the neighbor node is less than the cold-start delay of the container, namely $2\ell_{v,v',t} \leq d^n$ (Line 11). In this case, offloading requests to neighbor nodes $v' \in \mathcal{V}'$ is more beneficial than creating new containers in the current node. If $a^n_{v',t} \geq k^n_{v,t}$, it implies that there are sufficient cached containers on neighbor node v' to serve the remaining requests. Hence, the algorithm assigns the remaining request to node v' by $m^n_{v\to v',t} \leftarrow k^n_{v,t}$. Otherwise, if $a^n_{v',t} < k^n_{v,t}$, the algorithm assigns part of the requests to node v' and iterates over next neighbor node (Line 18–20).

If, after iterating over neighbor nodes, there still are unprocessed requests $k^n_{v,t} \neq 0$, new containers are created at the current node which will experience cold-start. If the current node cannot create new containers due to resource constraints, low-score containers are terminated using the proposed policy to free resources for new containers (Line 24–25).

Data locality: Data locality in edge computing refers to keep data close to where it is generated or used, rather than sending it to a distant edge node for processing. *FaasOrc* considers data locality in Algorithm 1 Line 2 where all nodes are sorted based on the distance to the request and we prioritize to process the data at the close edge node.

5.2. FaasOrc Beacon

At the node level, each edge node runs a Beacon instance, which periodically receives global information (i.e., invocation counts) from Commander. Beacon manages the lifetime of local functions and decides which one to evict based on the global information to reclaim hardware resources in the node. The cached functions have a configurable lifetime limitation $t_{threshold}$ (e.g., 5 min) and will be evicted by resource contentions (i.e., creating new function instances). By this

means, Beacon can avoid function "lock-in" which means keeping some popular functions alive for a long time. Meanwhile, Beacon periodically fetches information from Commander, such as the number of invocations for each function. This enables Beacon to avoid only caching locally popular functions and better handle the time-varying workloads. After finishing the execution, Beacon manages the life cycle of the function and decides whether to terminate it after execution.

Calculate function popularity score: Caching functions can mitigate the cold-start issue and avoid frequent creation and termination of containers. We use a lightweight and dynamic popularity score mechanism to identify hotspot functions (Eq. (10)).

The popularity score is derived from a web caching (Cherkasova, 1998) policy, which helps Beacon learn the characteristics of functions (i.e., historical invocation count, cold-start time and resource footprints). The function n popularity score at node v is represented by $H_v^n(t)$:

$$H_v^n(t) = t_v^n + \frac{\gamma_n \times d^n}{c_n}. \tag{10}$$

t_v^n refers to a "logical clock" to reflect the recency of local invocations. The clock is updated after every time interval (i.e., 1 min). When a container is used in edge node v, we assign the clock to the container and update the popularity score. Thus, containers that are used recently will have larger clock values and, hence higher scores. Also, t_v^n is node-specific and hence we can learn the local invocation characteristic at the node level.

γ_n is a global counter of a particular function invoked in a period of time (i.e., 30 min). After this period of time, we reset this counter to avoid "function lock-in", because hotspot function's counter can accumulate to a large value. γ_n helps to learn the global workload characteristics from the cluster level. The popularity score is proportional to the counter, and hence, more frequently invoked containers are kept alive for longer.

d^n refers to the cold start delay of a type n function. This parameter captures the benefit of caching a container. The popularity is proportional to the cold-start time of a container.

c_n refers to the memory allocation of a container in MB because cached containers are kept alive in memory. As the popularity score is inversely proportional to c_n, it is more expensive to keep larger containers alive instead of smaller containers.

Beacon combines global information (γ_n) and local information (t_v^n) to better handle workload skewness and oscillation. Beacon only comprises lightweight algorithms and hence it does not need any training before deployment.

Configure containers with a lifetime limitation: Beacon assigns each container a lifetime l_n which is configurable. After a function instance finishes execution, it will be automatically released after the lifetime. The rationale is that serverless invocations can oscillate over time. If the workload in an edge node drops significantly, it is important to terminate containers that have been not invoked for a period of time. Otherwise, high popular score functions will reside in the memory and waste resources.

Key algorithm in Beacon: Next, we introduce the key algorithm implemented in Beacon as shown in Algorithm 2.

Algorithm 2 shows the algorithm of creating new type n containers. If the residual resource capacity at node v is insufficient to create a new type n container, the algorithm terminates containers with the lowest popularity score in the cache until there are sufficient resources for creating a new type n container. If a container has been idle for more than a period of time $t_{threshold}$ (e.g., 5 min), it will also be destroyed to reclaim memory resources in the node, where l_n is the remaining lifetime of a function n. The reason is to avoid resource waste in light workloads, otherwise, some popular functions will exist in the memory for a long period of time.

Algorithm 2: Function management()

```
while k^n_{v,t} > 0 do
    if c_n > C^{rem}_v then
        Destroy the container with the lowest popularity score;
    else if l_n > t_threshold then
        Destroy n.
    else
        Create a new type n container;
        Update popularity score for the type n containers;
        k^n_{v,t} = k^n_{v,t} -1;
    end
end
```

5.3. Complexity analysis

The following theorem states that the performance of algorithm 1 is always bounded by $\max\{1+\frac{d^n}{p^n}, 1+2\frac{\ell_{v,v'}}{p^n}\}$.

Theorem 1. *The worst-case latency for handling a type n request is bounded by $\max\{1+\frac{d^n}{p^n}, 1+2\frac{\ell_{v,v'}}{p^n}\}$, where v denotes the edge node generating the request, and v' represents the edge node hosting the request.*

Proof. The offline optimal solution to process a type n request is distributing the request to the generated node with an available cached type n container. In this case, the link latency is zero as the request is placed on the node that generated the request, and the cold-start latency is also zero as the request uses a cached container. Hence, we obtain the lower bound for the latency of a single request as:

$$OPT(n) \geq p^n. \tag{11}$$

After that, we consider the maximum latency produced by the proposed algorithm. In the worst case, the current node has no cached or pre-warmed type n containers. The algorithm will offload a request to a neighbor node or create a new container at the current edge node. If a neighbor node meets the requirement that the link latency is less than the cold-start latency, the neighbor node processes the request by using a cached container. Otherwise, the current node creates a new type n container for the request at the cost of cold-start latency. We can define the end-to-end latency of the worst case as:

$$WORST(n) = \begin{cases} p^n + d^n & if \quad 2\ell_{v,v'} \geq d^n, \\ p^n + 2\ell_{v,v'} & if \quad 2\ell_{v,v'} < d^n. \end{cases} \tag{12}$$

We define $f(n) = WORST(n)/OPT(n)$ which represents the ratio of the worst case and best case. Hence, the maximal value of $f(n)$ denotes the worst-case competitive ratio. $f(n)$ can be represented as follows:

$$f(n) = \begin{cases} 1 + \frac{d^n}{p^n} & if \quad 2\ell_{v,v'} \geq d^n. \\ 1 + 2\frac{\ell_{v,v'}}{p^n} & if \quad 2\ell_{v,v'} < d^n, \end{cases} \tag{13}$$

Hence, the maximum $f(n)$ is given by $\max\{1+\frac{d^n}{p^n}, 1+2\frac{\ell_{v,v'}}{p^n}\}$. □

Theorem 2. *The time complexity of FaasOrc in every time step t is $\mathcal{O}(|\mathcal{V}|^2 \cdot |\mathcal{N}| \cdot \log(|\mathcal{V}|))$.*

Proof. Firstly, Algorithm 1 runs Line 1 $|\mathcal{N}|$ times which leads to a time complexity of $\mathcal{O}(|\mathcal{N}|)$.

Then, Algorithm 1 sorts all nodes, which has a time complexity of $\mathcal{O}(|\mathcal{V}| \cdot \log(|\mathcal{V}|))$. When offloading requests to neighbor nodes (Line 10), the algorithm iterates over the list of neighbor nodes with a time complexity of $\mathcal{O}(|\mathcal{V}|)$. For the assignments, Algorithm 2 may create containers and destroy others to make space for a small number of times and hence has a time complexity of $\mathcal{O}(1)$. Hence, handing N types of

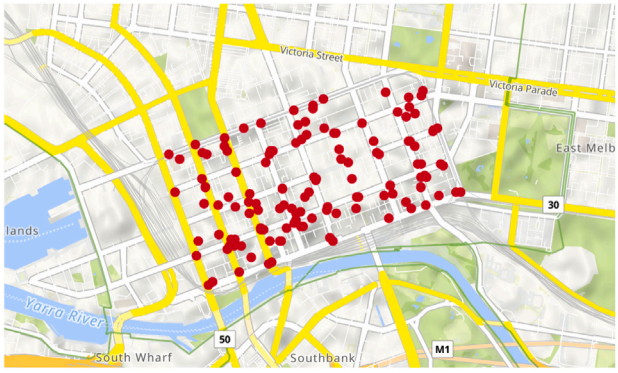


Fig. 4. The graphical map of edge servers in Melbourne CBD area.

requests over a network with V edge nodes has a time complexity of $\mathcal{O}(|\mathcal{V}| \cdot |\mathcal{N}| \cdot \log(|\mathcal{V}|))$. □

6. Performance evaluation

In this section, we conducted two sets of experiments with Network Simulator 3 (NS3) and a real-system prototype using **Knative** over Kubernetes, respectively.

6.1. Simulation setup

We first evaluate the performance of *FaasOrc* in a trace-driven simulation on a server with 105 GB RAM and an Intel(R) Xeon(R) E5645 processor with 24 cores. We use NS3 version 3.29, totaling around 2000 lines of code in C++. We have the following setups.

Topology: We use the widely used EUA dataset (Lai et al., 2018) which contains the information on edge servers in the Melbourne CBD area. The number of invocations ranges from hundreds of thousands of requests to a dozen. The topology consists of 125 edge nodes as shown in Fig. 4. The link latency is measured by the geographical distance between two edge nodes using their coordinates. The dataset exhibits a hybrid pattern, which is essential for modeling serverless workloads (A.Hasan et al., 2024).

Requests: As we cannot find any public serverless datasets for edge computing from major service providers, we leverage the Zipf-β popularity law to generate the request arrival pattern, which is widely used n edge computing (Pan et al., 2022; Shukla et al., 2018).

The Azure dataset (Azure, 2023) is used to create the request arrival pattern with the Zipf-β law. This dataset contains the invocations of functions on Microsoft Azure for 14 days. Each day is divided into 1440 time intervals and hence each time interval is 1 min. In each time interval, the requests can arrive and leave in arbitrary seconds.

We select 10 functions from the Azure dataset and the workload is generated using the Zipf distribution with the exponent β ranging from 0.5 to 1.5. A lower exponent β leads to greater disparity in the number of requests among edge nodes, increasing workload imbalance. The Zipf distribution is widely adopted to generate workloads in edge computing (Pan et al., 2022; Shukla et al., 2018; Shukla and Abouzeid, 2017).

Performance Benchmarks:

- ***FaasOrc*** adopts a cluster-level policy for request scheduling and a node-level policy for function management. At the cluster-level, we use Commander for load-balancing among multiple edge nodes. At the node-level, we use Beacon to handle function management.
- **Flame** (Yang et al., 2024) uses *Round Robin* for cluster-level request scheduling and an exponentially decaying algorithm to detect popular functions. More details can be found in Section 2.1.
- **FaasCache** (Fuerst and Sharma, 2021) also considers the heterogeneity of functions by using accumulative frequencies and cold-start overheads. It uses a local cache controller and has no lifetime limitations for functions. It also uses *Round Robin* for request scheduling. More details can be found in Section 2.3.
- **Hist** is a function caching algorithm proposed by Shahrad et al. (2020). Hist combines the Arimal model prediction and terminates the container after each function execution but pre-warms the container before a potential next invocation. As Hist does not discuss request scheduling, we implement Commander as its request scheduling policy.
- Fixed Caching (**FC**) is widely used in AWS (Lambda, 2023). It keeps a container alive for a fixed period of time. In the experiments, the caching period is set to 10 min. Similarly, we also implement Commander as its request scheduling policy.

6.2. Performance in simulation

Performance Summary:

Fig. 5 presents the results of the study across different workloads distributions in NS3 simulation. The response latency is the time between when users send a request and receive a response. The middle line represents the median value. The bottom and top of each "box" denote the 25th and 75th percentile, respectively. The top and bottom whiskers represent the 5th and 95 percentile. We observe that *FaasOrc* reduces the 75th percentile response latency by to 33.7%, 31.4%, 64% and 64.4% compared with Flame, FaasCache, Hist and FC. The reduction in latency to different workload stems from the use of orchestration (Commander schedules requests among different edge nodes and Beacon uses cluster information to manage container lifetime in each node). This design is vital as it is aware of cluster-level workload skewness and node-level oscillation simultaneously. For the 95th percentile of latency, *FaasOrc* receives 1.17 to 2.73 s while that of other benchmarks ranges from 1.84 to 3.78 s. The tail latency is dominated by the worst case where some functions experience cold starts and high communication latency.

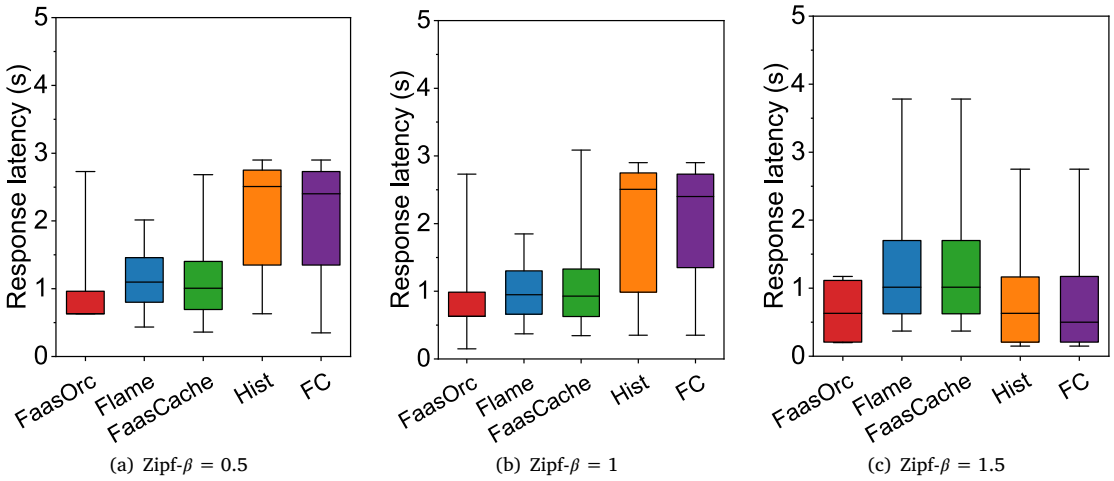
Fig. 6(a) shows the average response latency of all methods over different zipf-β values. We observe that *FaasOrc* receives the best performance, followed by Flame and FaasCache. The average response latency of *FaasOrc* ranges from 0.75 to 1.05 s, while that of Flame and FaasCache fluctuates between 1.08 and 1.49 s. This is because the combination of round-robin and function caching is insufficient for serverless edge computing. In other words, the workload among edge nodes and the popularity of functions must be considered simultaneously. TTL-based frameworks such as Hist and FC perform even worse as they adopt static caching policies.

Fig. 6(b) shows the cold-start frequency of different approaches. The cold-start frequency is the ratio of cold-start invocations and total invocations, which implies the occurrence of cold-start issues. The result demonstrates that the *FaasOrc* keeps the cold-start frequency between 0.04 and 0.14, while that of Flame and FaasCache fluctuates between 0.03 and 0.24. This is due to their local cache management causing hot function contention and imbalance between nodes. As a result, the overall performance in these methods fluctuates significantly.

FaasOrc performs the best in response latency under different workloads because it adopts a bi-level scheduling framework to learn distinct levels of system behavior in serverless edge computing. *FaasOrc* is designed to be workload agnostic, making it perform better under different workload distributions. Flame and FaasCache use round-robin load-balancing that spreads requests over different edge nodes even when the invocations are unevenly distributed.

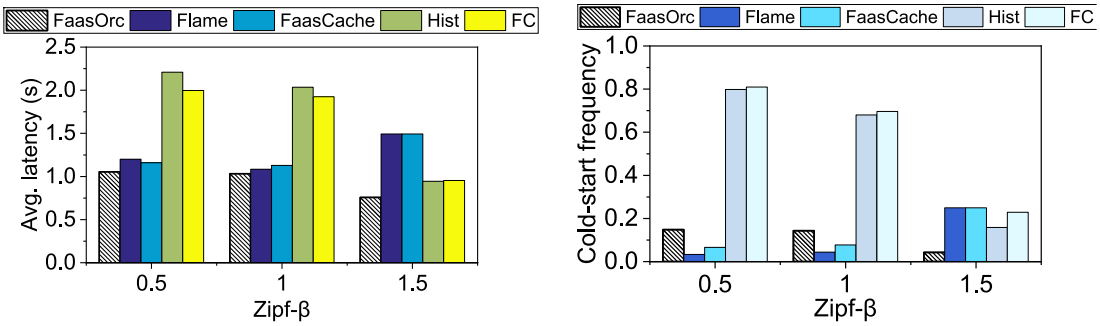
Impact of Parameter β:

β is the skewness parameter in the Zipf distribution that changes the distribution of workloads among different edge nodes. We use this parameter to examine the performance of this study under different



(a) Zipf-β = 0.5 (b) Zipf-β = 1 (c) Zipf-β = 1.5

Fig. 5. Box plots of response latency over different workloads in simulation. *FaasOrc* receives the best performance due to the incorporation of important serverless characteristics.



(a) Average latency (b) Cold-start frequency

Fig. 6. Average latency and cold-start frequency in simulation. *FaasOrc* efficiently reduces the cold-start occur and jointly considers network latency in edge computing.

workload distributions. When β increases, high-rank nodes will generate fewer requests and hence the distribution of requests is less skewed. As illustrated in Fig. 5, when β increases, the response of *FaasOrc* decreases due to its efficient request scheduling design. However, latency of round-robin-based approaches (Flame and FaasCache) increases because they do not aggregate functions in underutilized nodes. For TTL-based approaches (Hist and FC), their latencies also drop because Commander efficiently aggregates functions to edge nodes close to users under light workload (i.e., zipf-β = 1.5).

We observe a similar trend in Fig. 6(b) for cold-start frequency. When β increases from 0.5 to 1.5, *FaasOrc* achieves a lower cold-start frequency. This stems from Commander's policy to aggregate functions under light workloads. The cold-start frequency of Flame and FaasCache increases significantly. This is because round-robin distributes functions evenly and hence cached functions are insufficient to handle new requests under light workloads. Although Flame and FaasCache reduce cold-starts in heavy workloads, their average latency is still higher than that of *FaasOrc*. The rationale is that these methods may reuse cached containers in remote nodes first and hence suffer from high communication latency.

Average Latency with Different Workloads:

Fig. 6(a) shows the average latency over different factors of Zipf-β. *FaasOrc* receives the best performance of latency at 1.05 s, 1.03 s and 0.75 s, respectively. While not exciting, Flame and FaasCache yield average response latency ranging from 1.08 to 1.49 s. Hist achieves an average latency at 2.20 s, 2.03 and 0.94 s, respectively. FC receives an average latency of 1.99 s, 1.92 s and 0.95 s. When β = 1.5, Flame and FaasCache are beaten by TTL-based approaches. This improvement results from Commander's efficient request-scheduling design.

Cold-start Frequency with Different Workloads:

The cold-start frequency is the ratio of cold-start invocations and total invocations. This metric implies the occurrence of cold-start issues which have a significant impact on the response latency. As shown in Fig. 6(b), the cold-start frequency ranges from 0.04 to 0.14 under different workloads. In contrast, that of FaasCache and Flame ranges from 0.03 to 0.24 when Zipf-β is 0.5, 1.0 and 1.5, respectively. Flame and FaasCache perform better when β equals 0.5 and 1.0. This is not surprising because they may use remote edge nodes with less workloads first which ignores the trade-off between communication delay and cold-start delay. While not surprising, HIST and FC are outperformed by *FaasOrc* with cold-start frequencies ranging from 0.15 to 0.81.

FaasOrc efficiently reduces the occurrence of cold starts because it adaptively caches popular containers that are more likely to be invoked in the near future. Also, in the cluster level, *FaasOrc* schedules requests adaptively based on the incoming workloads. By this means, *FaasOrc* can effectively reduce the response latency by balancing request scheduling and function caching under different workload distributions.

6.3. Prototype in ***Knative***

To demonstrate that our framework can be readily deployed in real systems, we use the ***Knative*** implementation (Knative, 2023) which is

Table 2
Container instances.

Application name	Cold-start time	Memory size
Web Server	5.31 s	55 MB
File Processing	5.33 s	158 MB
Supermarket Checkout	5.34 s	332 MB
Image Recognition	4.89 s	92 MB

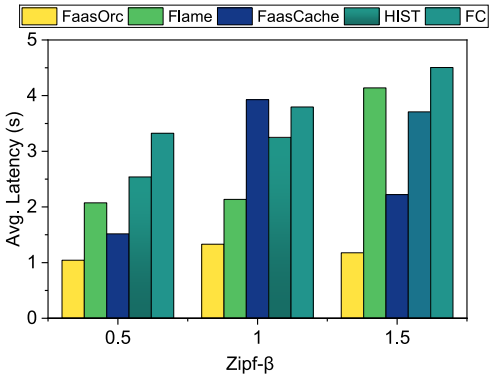


Fig. 7. The average latency in Knative. *FaasOrc* performs the best because it reduces both cold starts and network latency.

an enterprise-level solution for serverless applications. Knative is also used by Google Run Functions to provide modern serverless services.

Implementation Setup in Knative:

Our setup includes 1 controller node and 10 worker nodes. The controller node and worker nodes are servers equipped with 8 CPU cores and RAM in [16, 32] GB. The edge nodes span across different regions in Google Cloud, including *europe-west2 (London)*, *europe-west1 (Belgium)* and *europe-west4 (Netherlands)*. We have used Knative version 1.14.1 over Kubernetes 1.28, totaling 2200 lines of code in Go. Also, we reduced the number of invocations by 10000× to adapt to the system capacity. We first reduced the total number of requests in each time slot and then used the Zipf-β distribution to create workloads for the 10 worker nodes. We select four applications that are most frequently invoked and map them to four containers as shown in Table 2.

Containers:

In Knative, we build four types of containers derived from AWS Lambda functions as shown in Table 2. According to our measurement, the average cold-start time d^n lasts between 4.89 and 5.34 s, and the memory sizes are between 55 and 332 MB. The cold-start time is measured 10 times for each type of containers.

Performance Summary in Knative:

In Fig. 7, we observe that the *FaasOrc* reduces the average response latency by 49.7%, 31.1%, 59.0% and 68.7% compared to Flame, FaasCache, Hist and FC, respectively. The results of local cache policies (Flame and FaasCache) are far from achieving low latency due to skewed invocations across edge nodes.

The results in Knative justify that *FaasOrc* significantly improves the response latency due to (1) Scheduling requests from a global view of the cluster. (2) incorporating several critical characteristics of serverless invocations for caching.

Although the simulation and testbed implementation exhibit similar trends, the difference in the results is because the number of edge nodes and requests differs significantly between simulation and testbed.

Latency in Knative:

As illustrated in Fig. 7, we observe that *FaasOrc* achieves the best performance in terms of average response latency, ranging from 1.04 to 1.33 s. Flame achieves 2.07 to 4.13 s in terms of average latency. Similarly, that of FaasCache ranges from 1.51 to 3.92 s. This is because Flame and FaasCache make suboptimal caching decisions without jointly considering load balancing and function caching. The average latency of Hist ranges from 2.54 to 3.7 s. This is because Hist heavily relies on the distribution of histograms that exhibit a clear invocation pattern, where the pre-warm windows are ideally long and keep-alive windows are short. The average latency of FC is 3.32, 3.79 and 4.5 s, respectively. FC's performance is worse because it caches containers for a fixed period of time. FC does not consider function characteristics and cannot adapt to the frequently varying request pattern. Compared with the best-perform benchmarks, *FaasOrc* reduces the average latency by 31.1%.

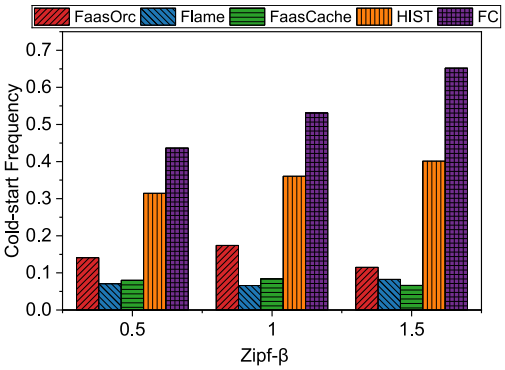


Fig. 8. The cold-start frequency in Knative. *FaasOrc* efficiently reduce cold starts due to the lightweight scheduling algorithm.

Cold-start Frequency in Knative:

Fig. 8 presents the cold-start frequency over different workloads. The cold-start frequency of *FaasOrc* is 0.14, 0.17 and 0.11, respectively. In contrast, the cold-start frequency of Flame and FaasCache ranges from 0.06 to 0.08 while that of Hist fluctuates between 0.314 and 0.401. Finally, FC is outperformed by other benchmarks due to its static policy where the cold-start frequency ranges from 0.436 to 0.651.

Flame and FaasCache slightly outperform *FaasOrc* because these methods may use remote edge nodes with lower workloads first, reducing cold-start occurrences but increasing communication delays.

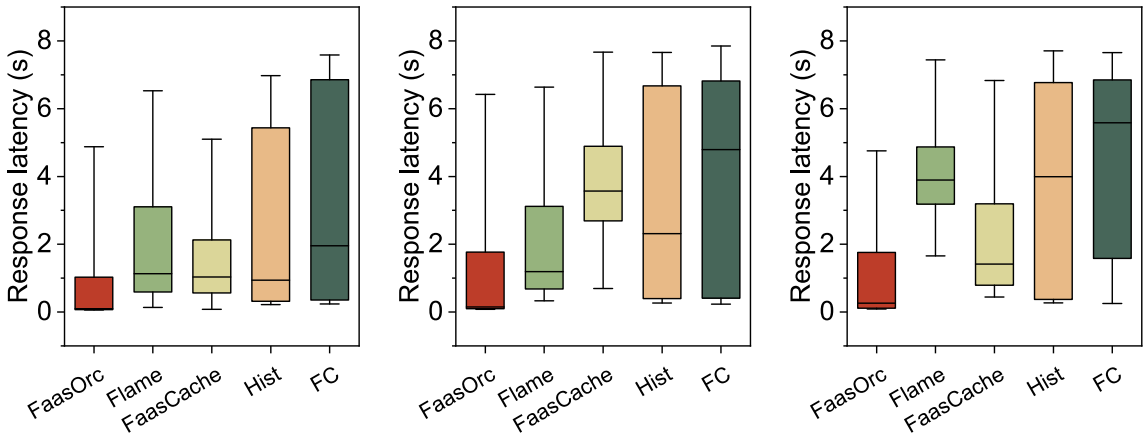
The Distributions of Latency in Knative:

In Fig. 9, we evaluate the performance of *FaasOrc* for different Zipf-β values. We use box plots to demonstrate the distributions of the response latency. The box plots show the 95th percentile, median and 5th percentile of the results. In Fig. 9(a), the median latency of *FaasOrc* is 0.097 s. The median latency is relatively low compared to average latency due to the fact that functions with cold starts in a remote node may experience high latency and hence significantly increase the average latency. In contrast, the median latency of Flame, FaasCache, Hist and FC are 1.13, 1.03, 0.94 and 1.95 s, respectively. This result demonstrates that the proposed algorithm dynamically adapts to skewed and time-varying workloads and thereby achieves better performance. Also, the 95th percentile of latency of *FaasOrc* is 4.87 s while that of Flame, FaasCache, Hist and FC are 6.52, 5.10, 6.97 and 7.58 s, respectively. This result indicates that *FaasOrc* also reduces the tail latency.

Fig. 9(b) presents the distributions of the latency for Zipf-β = 1. *FaasOrc* achieves the 75th percentile of latency at 1.77 s while that of Flame, FaasCache, Hist and FC are 3.11, 4.89, 6.67 and 6.81 s, respectively.

Fig. 9(c) shows the distributions of the latency for Zipf-β = 1.5. We observe that *FaasOrc* outperforms Flame, FaasCache, Hist and FC by up to 36.1%, 30.4%, 38.3% and 37.9% in terms of maximum latency.

When β increases from 0.5 to 1.5, *FaasOrc* shows a stable performance because Commander and Beacon complement each other, resulting in superior outcomes.



(a) Zipf-β = 0.5 (b) Zipf-β = 1 (c) Zipf-β = 1.5

Fig. 9. The box plots of latency in Knative. *FaasOrc* achieves the best performance due to the reduction in network latency and cold starts.

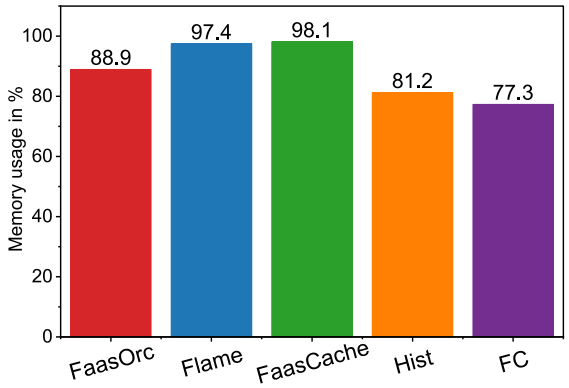


Fig. 10. Average memory usage of different benchmarks. *FaasOrc* receives moderate performance.

Fig. 10 shows the average memory usage of different benchmarks. FC receives the best performance as it is static and therefore does not adapt to the varying workloads. This can also be observed in latency where FC performs the worst. *FaasOrc* receives moderate performance due to the fact that *FaasOrc* dynamically adapts the number of cached containers so it uses more memory than static approaches like FC but significantly reduces cold starts. Compared to other dynamic methods like Flame and FaasCache, *FaasOrc* uses less memory and receives lower latency, further justifying its efficiency.

7. Conclusion and future work

In this paper, we studied the request scheduling and function management problem for serverless edge computing. We prove that the problem is NP-hard. Due to the computational complexity, we proposed a bi-level framework *FaasOrc* to reduce the response latency with a performance guarantee. *FaasOrc* schedules requests at the cluster level and manages functions based on a lightweight heuristic algorithm at the node level. The proposed *FaasOrc* algorithm is evaluated using real-world workload traces in simulation and a real-system prototype over **Knative**. *FaasOrc* efficiently takes advantage of global and local information, adapting to the dynamic and concurrent nature of serverless workloads. Our results show that the proposed algorithm reduces the response latency by 31.4% compared to the best-performing benchmark. The promising results imply that *FaasOrc* can be used for requests with service-level objects or QoS guarantee. However, limitations remain. Incorporating function caching may add extra overheads to the system and therefore the framework must be carefully designed. Also, it is worth including more edge nodes and function types in the testbed to justify the scalability. Moreover, network bandwidth and network topology can be critical performance bottlenecks due to the geographically distributed execution environments. Unlike centralized cloud infrastructures, edge platforms operate under limited and heterogeneous bandwidth constraints, while simultaneously serving latency-sensitive and bursty function invocations generated by end devices. The stateless and fine-grained nature of serverless functions further exacerbates bandwidth pressure by increasing control-plane traffic and data movement frequency. Consequently, effective congestion-aware scheduling, bandwidth-adaptive function placement, and lightweight communication protocols are essential to prevent performance degradation, ensure predictable latency, and maintain scalability in serverless edge systems.

This paper also raises a number of open questions that are interesting to investigate in the future. Future research directions include investigating the efficiency of *FaasOrc* in edge-cloud continuum. Due to the rapid growth of mobile edge computing, it is important to jointly consider the heterogeneity of cloud and edge clusters. Second, it would also be interesting to explore the resilience of serverless computing where functions and nodes can fail due to different reasons. For example, we can assume edge nodes may fail with a probability and investigate how to migrate services from one edge node to another. Finally, cross-region workload scheduling is another future direction. Due to the heterogeneity and network condition between different data centers, less congested regions can offer cheaper and stable options for executing jobs. It is therefore important for the research community to investigate the load balancing across data centers.

CRediT authorship contribution statement

Chen Chen: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Methodology, Investigation, Formal analysis, Conceptualization. **Lars Nagel:** Writing – review & editing, Supervision, Methodology, Investigation, Formal analysis, Conceptualization. **Lin Cui:** Methodology, Investigation, Conceptualization. **Weijia Jia:** Investigation, Conceptualization. **Fung Po Tso:** Writing – review & editing, Supervision, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

We know of no conflicts of interest associated with this publication, and there has been no significant financial support for this work that could have influenced its outcome. As the corresponding author, I confirm that the manuscript has been read and approved for submission by the named authors.

Acknowledgment

The work of Dr. Chen Chen was funded by the European Union under the project EDGELESS (GA no. 101092950).

Appendix. NP-hardness

Now we show that the Generalized Assignment Problem (GAP) (Chu and Beasley, 1997), which is known to be NP-hard, can be reduced to our problem in polynomial time. The GAP problem is assigning a number of K jobs to a set of J agents with minimum cost. Let w_j^k denote the amount of resource required by an agent j to perform job k. Also, d_j^k represents the cost of assigning job k to agent j. Let W_j denote the resource capacity of agent j and binary variable x_j^k represent whether job k is distributed to agent j. GAP can be formalized as follows.

Problem 2.

$$min \quad \sum_{j\in\mathcal{J}}\sum_{k\in\mathcal{K}} d_j^k x_j^k, \tag{A.1}$$

$$s.t. \quad \sum_{j\in\mathcal{J}} x_j^k = 1, \forall k, \tag{A.2}$$

$$\sum_{k=1}^{\mathcal{K}} w_j^k x_j^k \le W_j, \forall j, \tag{A.3}$$

$$x_j^k \in [0,1], \forall j, \forall k. \tag{A.4}$$

Now we show the equivalence of GAP and a simplified version of the proposed problem. If a type n request is only generated once in a node ($\sum_{v\in\mathcal{V}} m_{v,t}^n = 1$), the communication delay is 0 as we can distribute the request to the generated node. Also, we destroy the container immediately after serving a request. The proposed optimization problem becomes to minimize the latency $\sum_{v\in\mathcal{V}}\sum_{n\in\mathcal{N}} p^n m_{v,t}^n + d^n m_{v,t}^n$ while meeting the resource constraints $\sum_{n\in\mathcal{N}} c_n m_{v,t}^n \le C_v$. We formulate the simplified version of the latency optimization problem as follows:

Problem 3.

$$min \quad \sum_{v\in\mathcal{V}}\sum_{n\in\mathcal{N}} (p^n + d^n) m_{v,t}^n, \tag{A.5}$$

$$s.t. \quad \sum_{v\in\mathcal{V}} m_{v,t}^n = 1, \forall n, \tag{A.6}$$

$$\sum_{n\in\mathcal{N}} c_n m_{v,t}^n \le C_v, \forall v, \tag{A.7}$$

$$m_{v,t}^n \in [0,1], \forall n, \forall v. \tag{A.8}$$

If we map agents, jobs, job assignment cost d_j^k, the amount of required resources w_j^k and the resource capacity W_j in GAP to edge nodes, service requests, end-to-end latency, container resource demand c_n and resource capacity C_v, the equivalence of the **Problem 1** and **Problem 3** is achieved. Hence, the mapping can be done in a polynomial time.

Data availability

Data will be made available on request.

References

Agarwal, Siddharth, Rodriguez, Maria A., Buyya, Rajkumar, 2024. A deep recurrent-reinforcement learning method for intelligent AutoScaling of serverless functions. IEEE Trans. Serv. Comput. 17 (5), 1899–1910.

A.Hasan, Raed, Akawee, Mustafa M., Aziz, Enas F., Hammood, Omar A., Showket, Aws Saad, Jasim, Husniyah, Alkhafaji, Mohammed A., Ahmed, Omar K., Yasin, Khalil, 2024. Hybrid spotted Hyena based load balancing algorithm (HSHLB). Mesopotamian J. Big Data 2024, 199–210.

Akkus, Istemi Ekin, Chen, Ruichuan, Rimac, Ivica, Stein, Manuel, Satzke, Klaus, Beck, Andre, Aditya, Paarijaat, Hilt, Volker, 2018. SAND: Towards High-Performance serverless computing. In: 2018 USENIX Annual Technical Conference. USENIX ATC 18, USENIX Association, Boston, MA, pp. 923–935.

Alqaraghuli, Sarah Mohammed, Karan, Oguz, 2024. Using deep learning technology based energy-saving for software defined wireless sensor networks (SDWSN) framework. Babylon. J. Artif. Intell. 2024, 34–45.

Azure, 2023. AzurePublicDataset. https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md.

Azure Functions, 2023. Serverless functions in computing. https://azure.microsoft.com/en-gb/products/functions.

Cadden, James, Unger, Thomas, Awad, Yara, Dong, Han, Krieger, Orran, Appavoo, Jonathan, 2020. SEUSS: skip redundant paths to make serverless fast. In: Proceedings of the Fifteenth European Conference on Computer Systems. EuroSys '20, Association for Computing Machinery, New York, NY, USA.

Chai, Xiaohu, Zhou, Tianyu, Hu, Keyang, Tan, Jianfeng, Bie, Tiwei, Shen, Anqi, Shen, Dawei, Xing, Qi, Song, Shun, Yang, Tongkai, et al., 2025. Fork in the road: Reflections and optimizations for cold start latency in production serverless systems. In: 19th USENIX Symposium on Operating Systems Design and Implementation. OSDI 25, pp. 199–218.

Chen, Chen, Guan, Peiyuan, Chen, Ziru, Taherkordi, Amir, Hou, Fen, Cai, Lin X., 2025. Cost optimization for serverless edge computing with budget constraints using deep reinforcement learning. In: ICC 2025 - IEEE International Conference on Communications. pp. 5718–5723.

Cheng, Dazhao, Yan, Kai, Cai, Xinquan, Gong, Yili, Hu, Chuang, 2024. SLO-aware function placement for serverless workflows with layer-wise memory sharing. IEEE Trans. Parallel Distrib. Syst. 35 (6), 1074–1091.

Cherkasova, Ludmila, 1998. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. HP Labs Technical Report 98-69 (R.1).

Choi, Sang-Hoon, Park, Ki-Woong, 2022. Icontainer: Consecutive checkpointing with rapid resilience for immortal container-based services. J. Netw. Comput. Appl. 208, 103494.

Chu, P.C., Beasley, J.E., 1997. A genetic algorithm for the generalised assignment problem. Comput. Oper. Res. 24 (1), 17–23.

Emami Khansari, Mina, Sharifian, Saeed, 2025. A deep reinforcement learning approach towards distributed function as a service (FaaS) based edge application orchestration in cloud-edge continuum. J. Netw. Comput. Appl. 233, 104042.

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

Filinis, Nikos, Tzanettis, Ioannis, Spatharakis, Dimitrios, Fotopoulou, Eleni, Dimolitsas, Ioannis, Zafeiropoulos, Anastasios, Vassilakis, Constantinos, Papavassiliou, Symeon, 2024. Intent-driven orchestration of serverless applications in the computing continuum. Future Gener. Comput. Syst. 154, 72–86.

Fuerst, Alexander, Sharma, Prateek, 2021. 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. New York, NY, USA, pp. 386–400.

Gao, Bin, Zhou, Zhi, Liu, Fangming, Xu, Fei, 2019. Winning at the starting line: Joint network selection and service placement for mobile edge computing. In: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications. pp. 1459–1467.

Google Cloud, 2023a. Google functions. https://cloud.google.com/functions.

Google Cloud, 2023b. Implementing SLOs. https://sre.google/workbook/implementing-slos/.

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

Gu, Lin, Zeng, Deze, Hu, Jie, Jin, Hai, Guo, Song, Zomaya, Albert Y., 2021. Exploring layered container structure for cost efficient microservice deployment. In: IEEE INFOCOM 2021 - IEEE Conference on Computer Communications. pp. 1–9.

Hasan, Raed A., Yasin, Khalil, Kareem, Parween R., Salih, Ali Mohammed, Jasim, Husniyah, Ameedudeen, Mohamed Arif, Abbas, Mohammad Abdullah, Rachini, Ali, oirere, Aaron Mogeni, 2025. Energy-efficient task offloading and resource allocation in mobile cloud computing using edge-AI and network virtualization. KHWARIZMIA 2025, 42–49.

He, Ting, Khamfroush, Hana, Wang, Shiqiang, La Porta, Tom, Stein, Sebastian, 2018. It's hard to share: Joint service placement and request scheduling in edge clouds with sharable and non-sharable resources. In: 2018 IEEE 38th International Conference on Distributed Computing Systems. ICDCS, pp. 365–375.

Joosen, Artjom, Hassan, Ahmed, Asenov, Martin, Singh, Rajkarn, Darlow, Luke, Wang, Jianfeng, Deng, Qiwen, Barker, Adam, 2025. Serverless cold starts and where to find them. In: Proceedings of the Twentieth European Conference on Computer Systems. EuroSys '25, Association for Computing Machinery, New York, NY, USA, pp. 938–953.

Karamzadeh, Amirmohammad, Shameli-Sendi, Alireza, 2024. Reducing cold start delay in serverless computing using lightweight virtual machines. J. Netw. Comput. Appl. 232, 104030.

Knative, 2023. Home. https://knative.dev/docs/.

Lafta, Noor Abdalkareem, Al-fiskhaltom, Fadi Raafat Fadheel, 2023. A comprehensive survey of dynamic hashing techniques in network data processing. Babylon. J. Netw. 2023, 89–93.

Lai, Phu, He, Qiang, Abdelrazek, Mohamed, Chen, Feifei, Hosking, John, Grundy, John, Yang, Yun, 2018. Optimal edge user allocation in edge computing with variable sized vector bin packing. In: Service-Oriented Computing. Springer International Publishing, pp. 230–245.

Lambda, A.W.S., 2023. Amazon web services. https://aws.amazon.com/lambda/.

Lee, Geonsun, Yang, Yue, Healey, Jennifer, Manocha, Dinesh, 2025. Since U been gone: Augmenting context-aware transcriptions for re-engaging in immersive VR meetings. In: Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems. CHI '25, Association for Computing Machinery, New York, NY, USA.

Li, Suyi, Wang, Wei, Yang, Jun, Chen, Guangzhen, Lu, Daohe, 2023. Golgi: Performance-aware, resource-efficient function scheduling for serverless computing. In: Proceedings of the 2023 ACM Symposium on Cloud Computing. SoCC '23, Association for Computing Machinery, New York, NY, USA, pp. 32–47.

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

Lin, Yanying, Li, Yanbo, Peng, Shijie, Tang, Yingfei, Luo, Shutian, Shen, Haiying, Xu, Chengzhong, Ye, Kejiang, 2024. QUART: Latency-aware faas system for pipelining large model inference. In: 2024 IEEE 44th International Conference on Distributed Computing Systems. ICDCS, pp. 1–12.

Liu, Junhan, Cai, Zinuo, Liu, Yumou, Li, Hao, Zhang, Zongpu, Ma, Ruhui, Buyya, Rajkumar, 2025. SMore: Enhancing GPU utilization in deep learning clusters by serverless-based co-location scheduling. IEEE Trans. Parallel Distrib. Syst. 36 (5), 903–917.

Mahgoub, Ashraf, Yi, Edgardo Barsallo, Shankar, Karthick, Elnikety, Sameh, Chaterji, Somali, Bagchi, Saurabh, 2022. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). USENIX Association, Carlsbad, CA, pp. 303–320.

Moakhar, Sahar Pilevar, Abrishami, Saeid, 2024. An efficient mechanism for function scheduling and placement in function as a service edge environment. J. Netw. Comput. Appl. 226, 103890.

nsnam, 2025. Ns-3 | a discrete-event network simulator for internet systems.

Pan, Li, Wang, Lin, Chen, Shutong, Liu, Fangming, 2022. Retention-aware container caching for serverless edge computing. In: IEEE INFOCOM 2022 - IEEE Conference on Computer Communications. pp. 1069–1078.

Pandey, Manish, Tak, Byungchul, Kwon, Young-Woo, 2025. PROBA: Enhancing serverless edge computing via adaptive task scheduling and probabilistic resource sharing. In: 2025 IEEE 18th International Conference on Cloud Computing. CLOUD, pp. 351–361.

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

Pmdarima, 2025. Alkaline-ml/pmdarima: A statistical library designed to fill the void in python's time series analysis capabilities.

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

Qi, Shixiong, Monis, Leslie, Zeng, Ziteng, Wang, Ian-chin, Ramakrishnan, K.K., 2022. 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, Association for Computing Machinery, New York, NY, USA, pp. 780–794.

Rangarajan, Sarathkumar, Al-Quraishi, Tahsien, 2023. Navigating the future of the internet of things: Emerging trends and transformative applications. Babylon. J. Internet Things 2023, 8–12.

Roy, Rohan Basu, Patel, Tirthak, Tiwari, Devesh, 2022. 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, New York, NY, USA, pp. 753–767.

Russo Russo, Gabriele, Cardellini, Valeria, Lo Presti, Francesco, 2024. A framework for offloading and migration of serverless functions in the edge–cloud continuum. Pervasive Mob. Comput. 100, 101915.

Sakirin, Tam, Asif, Iqra, 2023. Infusing k-means for securing IoT services in edge computing. Mesopotamian J. Comput. Sci. 2023, 39–46.

Shahrad, M., Fonseca, R., Goiri, Íñ., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., Bianchini, R., 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In: Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference.

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

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

Shujairi, Murtadha, 2025. Developing IoT performance in healthcare through the integration of machine learning and Software-Defined Networking (SDN). Babylon. J. Internet Things 2025, 77–88.

Shukla, Samta, Abouzeid, Alhussein A., 2017. Proactive retention aware caching. In: IEEE INFOCOM 2017 - IEEE Conference on Computer Communications. pp. 1–9.

Shukla, Samta, Bhardwaj, Onkar, Abouzeid, Alhussein A., Salonidis, Theodoros, He, Ting, 2018. Proactive retention-aware caching with multi-path routing for wireless edge networks. IEEE J. Sel. Areas Commun. 36 (6), 1286–1299.

Tütüncüoğlu, Feridun, Ben-Ameur, Ayoub, Dán, György, Araldo, Andrea, Chahed, Tijani, 2024. Dynamic time-of-use pricing for serverless edge computing with generalized hidden parameter Markov decision processes. In: 2024 IEEE 44th International Conference on Distributed Computing Systems. ICDCS, pp. 668–679.

Wang, Fei, Jiao, Lei, Zhu, Konglin, Lin, Xiaojun, Li, Lei, 2023. Toward sustainable AI: Federated learning demand response in cloud-edge systems via auctions. In: IEEE INFOCOM 2023 - IEEE Conference on Computer Communications. pp. 1–10.

Wang, Ziyi, Zhang, Songyu, Cheng, Jing, Wu, Zhixiong, Cao, Zhen, Cui, Yong, 2023. Edge-assisted adaptive configuration for serverless-based video analytics. In: 2023 IEEE 43rd International Conference on Distributed Computing Systems. ICDCS, pp. 248–258.

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

Yang, Yue, Leuze, Christoph, Hargreaves, Brian, Daniel, Bruce, Baik, Fred, 2025. EasyREG: Easy depth-based markerless registration and tracking using augmented reality device for surgical guidance. arXiv:2504.09498.

Yang, Yanan, Zhao, Laiping, Li, Yiming, Wu, Shihao, Hao, Yuechan, Ma, Yuchi, Li, Keqiu, 2024. Flame: A centralized cache controller for serverless computing. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4. ASPLOS '23, Association for Computing Machinery, New York, NY, USA, pp. 153–168.

Yang, Yanan, Zhao, Laiping, Li, Yiming, Zhang, Huanyu, Li, Jie, Zhao, Mingyang, Chen, Xingzhen, Li, Keqiu, 2022. INFless: a native serverless system for low-latency, high-throughput inference. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '22, Association for Computing Machinery, New York, NY, USA, pp. 768–781.

Yu, Hanfei, Basu Roy, Rohan, Fontenot, Christian, Tiwari, Devesh, Li, Jian, Zhang, Hong, Wang, Hao, Park, Seung-Jong, 2024. RainbowCake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. ASPLOS '24, Association for Computing Machinery, New York, NY, USA, pp. 335–350.

Yu, Minchen, Cao, Tingjia, Wang, Wei, Chen, Ruichuan, 2023. Following the data, not the function: Rethinking function orchestration in serverless computing. In: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). USENIX Association, Boston, MA, pp. 1489–1504.

Yue, Xiaofei, Yang, Song, Zhu, Liehuang, Trajanovski, Stojan, Fu, Xiaoming, 2024. Demeter: Fine-grained function orchestration for geo-distributed serverless analytics. In: IEEE INFOCOM 2024 - IEEE Conference on Computer Communications. pp. 2498–2507.

Zhang, Yuqiu, Jacobsen, Hans-Arno, 2025. Mocha: Scalable and compliant function scheduling for federated serverless computing. In: Proceedings of the 26th International Middleware Conference. Middleware '25, Association for Computing Machinery, New York, NY, USA, pp. 398–412.

Zhang, Yining, Jiao, Lei, Zhu, Konglin, Lin, Xiaojun, Zhang, Lin, 2025. Toward market-assisted AI: Cloud inference for streamed data via model ensembles from auctions. IEEE Trans. Netw. 33 (2), 793–806.

Zhou, Zhuangzhuang, Zhang, Yanqi, Delimitrou, Christina, 2022. AQUATOPE: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. In: ASPLOS 2023, Association for Computing Machinery, New York, NY, USA, pp. 1–14.