Improving the Serverless Function Cache Efficiency With Flame

Yanan Yang, Wenda Tang, Laiping Zhao, Keqiu Li, *Fellow, IEEE*, and Jie Wu, *Fellow, IEEE*

***Abstract*—Function caching is one of the fundamental techniques in FaaS platforms to alleviate coldstart overhead. However, as cache instances consume significant cloud resources (e.g., memory), it is challenging to balance function performance and cache cost. Current systems use simple and rudderless cache polices with a "local cache control" design, which ignores function characteristics such as workload skewness from hot functions and results in either cache contentions or cache resource waste. In this paper, inspired by software-defined networks, we propose `Flame`, an efficient cache system to manage cached functions with hotspot-aware instance scheduling and cache allocation. It consists of a two-layer design. Firstly, by decoupling the cache control plane from worker nodes and introducing a centralized cache controller, `Flame` can schedule functions from a global view of the cluster's status, thereby reducing inter-node workload skew. Second, `Flame` divides the prior monolithic cache pool within each node into multiple partitions and dynamically assigns them to different hot functions, thereby further mitigating intra-node cache contention. Experimental results from real-world workloads show that `Flame` can reduce cache resource usage by 36% on average while improving function performance by nearly 7× compared to the state-of-the-art method.**

***Index Terms*—Serverless, coldstart overhead, function caching.**

I. Introduction

FUNCTIONS-AS-A-SERVICE (FaaS) serverless platforms provide a stateless programming model for cloud users, which allows developers to construct applications as a set of event-triggered functions without considering environment or resource management (e.g., auto-scaling). Due to the stateless nature of FaaS functions, user requests may suffer from a coldstart problem (i.e., launching a function instance from scratch from a microVM or container) when an occasional request arrives. When coldstart occurs, the startup time of a new function instance can be orders of magnitude higher than the execution time [1], making it one of the most pressing performance issues in serverless computing [2], [3], [4], [5].

Function caching is a widely used approach in today's FaaS platforms to mitigate coldstart overhead, enabling functions to be invoked in a "warm-start" manner. By caching function instances after each invocation rather than reclaiming them immediately, the subsequent requests can reuse them to avoid coldstart invocation (as recovering a paused container takes only 0.5ms [6]). However, caching functions also increase cloud cost. For example, one of China's largest cloud providers consumes more than 20% of cluster memory for keeping functions alive under a $<1\%$ of coldstart ratio. Since it is almost impossible to cache all functions in the FaaS platform, serverless providers commonly launch a *local cache controller* on each worker (a VM node or physical server that runs FaaS functions) to dynamically manage the lifetime of cached instances using TTL-based (Time-to-live) [2], [7] or priority-based caching policies [5], [8].

Unfortunately, we find that cache contention is widespread in current systems and hinders high function cache efficiency. There are two main reasons. On the one hand, analysis of Azure's function trace [3] shows that 90% of invocations come from only 10% of hot functions. *Local cache controller* cannot cooperate with instance scheduling, and workload skewness from hot functions can lead to severely imbalanced inter-worker cache usage. The lack of a global view of workload characterization also makes it difficult to make optimal cache decisions. Hence, workers with aggregated hot functions may suffer from cache congestion, while others may experience significant cache resource waste. Our analysis shows that more than 50% of cached instances are rarely invoked in such a design, and the local hotspot contention can lead to 38% of coldstart ratio fluctuations, degrading both cache efficiency and function performance.

On the other hand, existing FaaS platforms predominantly use a monolithic cache pool per worker, allowing local functions to share the same cache resources, and this inherently leads to cache contention. While some priority-based caching policies tend to reduce cache allocation for cold functions, thereby alleviating intra-worker cache contention [5], resource conflicts across hot functions persist. Our analysis shows that even functions with similar popularity can exhibit significant

Received 21 April 2025; revised 31 October 2025; accepted 7 December 2025. Date of publication 12 October 2025; date of current version 12 February 2026. This work was supported in part by Shandong Provincial Natural Science Foundation Project under Grant ZR2022LZH018, in part by the National Natural Science Foundation of China under Grant 62372322, Grant 62432015, and Grant U25B2021, and in part by Tianjin Science and Technology Plan Project under Grant 24ZXKJGX00060. Recommended for acceptance by H. Jiang. *(Corresponding author: Laiping Zhao.)*

Yanan Yang, Wenda Tang, and Jie Wu are with China Telecom Cloud Computing Research Institute, Beijing 1000832, China (e-mail: yangyn11@chinateleccom.cn; tangwd1@chinatelecom.cn; wujie@chinatelecom.cn).

Laiping Zhao and Keqiu Li are with the College of Intelligence and Computing, Tianjin University, Tianjin 300354, China (e-mail: laiping@tju.edu.cn; keqiu@tju.edu.cn).

Digital Object Identifier 10.1109/TC.2025.3642338

0018-9340 © 2025 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.

performance disparities. For functions from different cloud tenants, this would compromise fairness. For functions within an application workflow, this can lead to several times the end-to-end latency variation.

In this paper, inspired by the studies on software-defined networks, we argue for a hotspot-aware function caching system to make efficient cache decisions in FaaS platform, which can address the inter-worker workload skewness and intra-worker cache contention from a two-layer controller design: Firstly, by decoupling the control plane from local worker and introducing a *centralized cache controller*, caching decisions can be made to incorporate with cluster-level instance scheduling, enabling better workload balancing and reducing the local cache contentions. Additionally, on each worker, by using a partition-based cache policy instead of existing monolithic cache pool design, functions can be mapping to different partitions for better cache allocation, and the intra-worker cache conflicts can be improved.

There are several challenges that need to be addressed. The cache controller should answer the *"4H"* questions: *(1) How to design a hotspot-aware cache scheduling algorithm to alleviate workload skewness?* Simple random or round-robin request dispatching is not sufficient and will hurt locality. *(2) How to choose the functions that need to be cached to maximize cache efficiency?* As the cache resources on each worker are limited and variable. *(3) How to determine the partition size for each function to reduce hotspot contention?* A static cache allocation can not adapt to workload changes and will cause great performance disparity. Meanwhile, the cache system relies closely on workload and cluster status monitoring. *(4) How to reduce the system runtime and communication overhead*, and make it easy to integrate into existing FaaS platforms with good scalability, is also a challenge.

We propose **`Flame`** to overcome these challenges. **`Flame`** consists of a top *Cache controller* and multiple agents (named *Cachelet*) deployed on each worker. The *Cachelet* collects local worker status and workload information, interacts with the cache controller, and executes cache decisions. **`Flame`** introduces the concept of a "hotspot" for both functions and workers, which can describe the popularity of a function and the load pressure on workers, respectively. The *Cache controller* dynamically identifies the global hot functions based on their invocation counts and schedules them with a *minimum hotspot aggregation* algorithm to alleviate workload skewness. On each worker, hot functions are cached across several protected memory partitions to improve performance. In contrast, non-hot functions are cached in a "best-effort" manner to utilize the worker's idle resources. The *Cachelet* uses a heuristic re-partitioning algorithm that dynamically adjusts the partition size of each hot function based on its popularity and performance changes, thereby achieving high cache efficiency and better fairness.

We implement **`Flame`**'s cache control on OpenFaaS [9], a widely adopted open-source FaaS framework based on Kubernetes. To evaluate the effectiveness and cache efficiency of **`Flame`**, we utilize real-world benchmarks from ServerlessBench [10] and FunctionBench [11]. Experimental results demonstrate that **`Flame`** reduces cache costs by an average of 36%, achieves a warm invocation ratio exceeding 99.3% in FaaS clusters, and improves function performance by nearly 7× compared to existing function cache systems.

Our contributions can be summarized as follows:

- We present several observations and a detailed analysis of the low cache efficiency problem in current "local cache control" systems on FaaS platforms.
- We implement our "centralized cache control" proposal in **`Flame`**, an efficient cache management system for FaaS functions. It identifies the cluster-level hot functions and alleviates hotspot contention caused by workload skewness through hotspot-aware cache scheduling.
- We also propose a hybrid caching policy with a partition-based cache pool per worker. By assigning functions to different cache partitions and dynamically tuning their sizes, it can further improve the cache redundancy and intra-worker hotspot contention.
- We present a full-system implementation of **`Flame`** on OpenFaaS, and our experimental results on real-world workloads demonstrate its high cache efficiency and better function performance.

II. Motivation

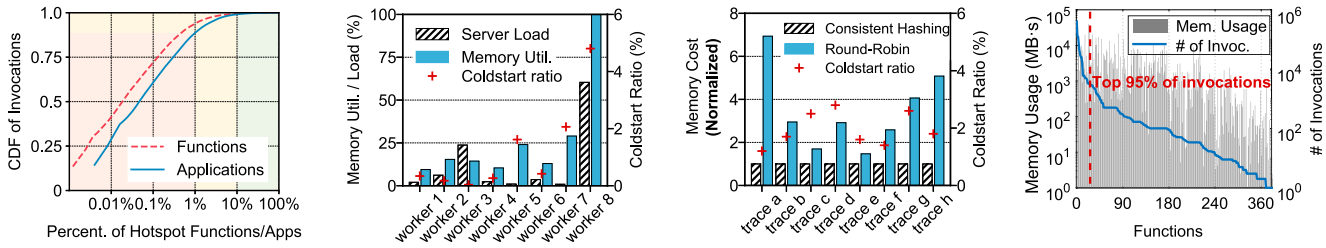
The stateless nature of FaaS functions allows serverless platforms to automatically adjust the number of function instances as the workload changes. Typically, a FaaS cluster may contain hundreds of workers serving tens of millions of requests from thousands of functions. For each function, the gateway receives user requests and forwards them to workers with idle function instances for processing; if none are available, a new instance is created. As launching functions from scratch incurs 100s to 1000s of ms of coldstart latency, depending on user code size and runtime languages [3], [12], a common practice among cloud providers and FaaS platforms is to employ cache policies to keep functions "warm". For example, AWS Lambda caches function instances for about 15−60 minutes after an invocation finishes. FaasCache system proposes a priority-based cache policy at each worker to keep functions warm [5]. These function caching mechanisms can alleviate the coldstart overhead while balancing cache overhead (e.g., memory consumption).

Hot functions are widely used in serverless scenarios, leading to workload skewness as a common phenomenon in FaaS platforms. For example, more than 90% of requests in Azure come from approximately 10% of hot functions (Fig. 1(a)). Given that FaaS front-end gateways always use load-balancing rules, such as consistent hashing, to forward a function's requests to the last-accessed workers, this "server lock-in" can easily result in workload skew across workers. When multiple hot functions are colocated on local workers, server pressure can increase dramatically, leading to intra-worker cache contention and degraded function performance. In the following, we give several observations to explain why the existing function cache system is insufficient to achieve high resource efficiency.

A. Limitations of Local Cache Control

We reproduce FaasCache, which is a state-of-the-art local cache control system [5], on an 8-worker FaaS cluster. Each

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.



(a) Function call statistics (b) Hotspot contention (c) Low resource efficiency (d) Cache redundancy

Fig. 1. (a) Characteristics of hot functions in Azure's function trace. (b) Cache contentions due to workload skewness under local cache control in FaaS cluster. (c) Cache contention and function performance comparison under different workload distributions (hashing-based load balancing and round-robin load balancing) with local cache control. (4) Cache redundancy under local cache control.

worker is configured with 128 GB of memory. To collect the experimental results of FaasCache, we deploy several real-world functions and generate requests following Azure's function traces. The front-end gateway includes consistent hashing and round-robin load-balancing rules to observe cache behavior.

Observation #1: Workload skewness: Local cache control is unable to handle workload skewness in FaaS cluster. Hotspot contention can result in a 100× difference in function performance across workers.

Fig. 1(b) illustrates the server load, memory utilization, and coldstart ratio on each worker under the hashing-based forwarding rule, where the server load denotes the proportion of processed requests relative to the total workload. It is evident that worker eight experiences the highest server load and memory usage because over 50% of hot function invocations are routed to it for processing, whereas worker 3 handles approximately 20% of the total requests from hot functions. This suggests that cache contention on worker 8 is significantly more severe in the cluster, resulting in its coldstart ratio being 100 × higher than that of other workers (4.7% on worker 8 compared to 0.04% on worker 3).

Eliminating workload skewness can improve resource efficiency under local cache control. As shown in Fig. 1(c), switching the load-balancing rule to round-robin enables more even distribution of hot functions in FaaS cluster, thereby eliminating performance bottlenecks on local workers (e.g., worker 8); however, this may reduce cache locality, leading to either higher coldstart ratios (1.8%–3.9% across workers) or 3× of higher memory cost to maintain a low cache miss ratio.

Observation #2: Cache Redundancy: Local cache control results in substantial cache redundancy, with over 75% of cache resources being wasted within the cluster.

Local cache control allows each worker to make independent cache decisions. For each function, its instances may be redundantly cached across multiple workers, leading to inefficient use of cache resources. To analyze cache utilization, we collected memory usage data from 384 real-world functions over 1 day of invocations. The memory usage of each FaaS function includes both the memory allocated for caching its instances and the memory consumed during execution requests. Fig. 1(d) presents the details, with functions sorted in descending order by the number of invocations. The top-20 most frequently invoked functions account for nearly 95% of all invocations. However, these functions consume less than 20% of the total memory usage across the workload. This implies that more than 80% of the cluster's memory usage is devoted to caching non-hot functions, which are rarely invoked.

We further analyzed the memory usage breakdown of the remaining 365 non-hot functions and found that only 5% of the memory is used for execution, indicating significant cache redundancy and low cache utilization (ranging from 10% to 30% per cached instance). If the caching system could identify over-provisioned function instances and reclaim them through a global cache management perspective, the cache cost could be substantially reduced.

B. Limitations of Monolithic Cache Pool

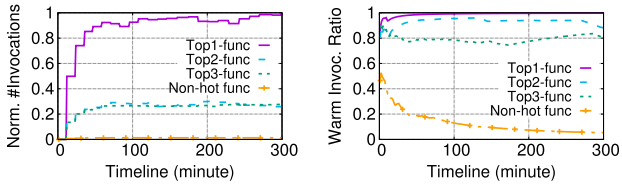
Previous observations have revealed cache contention due to workload skew under local cache control. However, using only cluster-level hotspot-aware scheduling is insufficient, as the monolithic worker cache pool design inherently leads to intra-worker hotspot contentions. Suboptimal cache allocation among hot functions on the local worker may result in performance fluctuations and tenant unfairness, and changes in workload (e.g., function popularity evolving) will greatly complicate cache management.

In the following, we present several observations that reveal the inadequacy of existing monolithic cache-pool architectures on each worker.

Observations #3: Intra-worker contention: Hotspot-aware instance scheduling is insufficient to eliminate intra-worker cache contention, leading to nearly 20% of coldstart ratio disparity.

We conduct experiments from a modified version of the FaasCache system, where functions are manually categorized into hotspot and non-hotspot types, and scheduled to a worker with minimum server load. Functions on each worker are cached following FaasCache's priority-based keep-alive policy. Fig. 2(a) shows the runtime behavior of four representative functions on one worker (function profiles are listed in Table I). We can see that *Top1-func* has the highest request arrival rate and the greatest popularity. In contrast, the workload characterization and popularity are very similar between *Top2-func* and *Top3-func*. This is typical on a FaaS platform, as upstream and downstream

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.



(a) Function popularity (b) Function performance

Fig. 2. Intra-worker cache contentions on monolithic worker cache pool design: (a) function popularity (# of invocations) of three hot functions and a baseline function; (b) function performance disparity of the above functions.

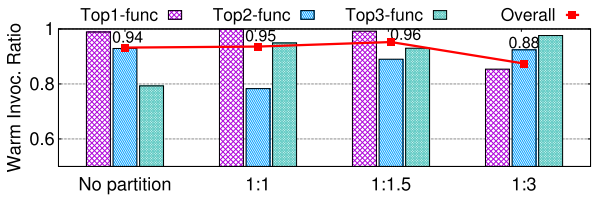


Fig. 3. Function performance under cache partitions. The worker cache Pool is divided into two partitions, and *Top-func1* is allocated a separate partition. We examine the sizes of the two partitions from 1:1, 1:1.5, and 1:3, respectively.

TABLE I
FOUR REPRESENTATIVE FUNCTIONS USED IN MOTIVATION EXPERIMENTS

Id	Func. name	Function Description	# of Invocation	Requests/s
#1	Top1-func	key-downloader_512m	1,255,215	14.5 reqs/s
#2	Top2-func	parallel-alu-mp_512m	363,326	4.2 reqs/s
#3	Top3-func	extract-image_512m	353,487	4.1 reqs/s
#4	Non-hot func	thumbnail_256m	17,288	0.2 reqs/s

functions within a workflow may have similar access patterns and arrival rates. We also show a non-hot function as a baseline.

Fig. 2(b) further illustrates the warm invocation ratio (1 - coldstart rate) of these four functions. It is evident that Top1-func demonstrates the best performance. Although *Top2-func* and *Top3-func* share more than 97% similarity in workload access patterns, there is a notable performance disparity between them. Specifically, *Top2-func* and *Top3-func* exhibit 92% and 79% of average warm invocation ratios, respectively. Analysis of cache usage within workers reveals the main reasons. First, *Top1-func*'s high frequency of invocations preempts a significant portion of cache resources, putting much pressure on *Top2-func* and *Top3-func*. At the same time, short-term workload fluctuations between them result in oscillations in hotspot decisions, which degrade the performance of *Top3-func*.

Observations #4: Cache partition: Cache partitioning can fundamentally help mitigate inter-worker hotspot contention, and improve function performance across different tenants without increasing overall cache cost.

Inspired by prior studies that used partition-based techniques to isolate cloud resources (e.g., CPU cores) across colocated workloads, cache partition can also be used in a FaaS platform to address intra-worker hotspot contention. For this purpose, we keep the testbed environment in Fig. 2 and divide the worker cache pool into two parts, where the first partition is allocated to *Top1-func*, while the second partition is assigned to the rest of the functions. The cache partition sizes follow the ratios 1:1, 1:1.5, and 1:3, respectively. We collect experimental results from both the cache-partition and no-partition groups, using metrics consisting of the function warm-up invocation ratio (i.e., cache hit) and its geometric mean.

Fig. 3 illustrates the evaluation results, showing that tenants' function performance varies with the partitioning size compared to the non-partitioned group. The 1:1.5 partition group achieves the best performance, which not only maintains the high performance of *Top1-func* but also improves the cache hit ratios of both *Top2-func* and *Top3-func* with a smaller gap between their performance. Given that FaaS clusters typically have only a small number of hot functions, it is feasible to allocate separate cache partitions for them, thereby reducing cache contention per worker.

Moreover, it is evident that different cache partitions can affect overall function performance. For instance, the cache contention between *Top2-func* and *Top3-func* becomes even worse under the 1:1 group, resulting in nearly 15% of cache misses increasing for *Top2-func*, while the 1:3 cache partition reversely causes performance degradation for the *Top1-func*. Thus, determining an appropriate partition policy is also challenging.

C. Implications

The above observations motivate us to design a novel and efficient function cache system for the FaaS platform. To overcome the shortcomings of local cache control, the cache system should be able to identify hot functions and schedule them from a global view of cluster status, thus alleviating hotspot contentions from workload skewness (Observation 1). Additionally, the caching decisions for each FaaS function should adaptively tune to workload and worker status changes, thereby reducing inter-worker cache redundancy and improving cache efficiency (Observation 2). On each worker, hot functions should be carefully isolated with appropriate cache resource allocation, thereby reducing the oscillation of cache decisions and performance fluctuations (Observations 3&4). Finally, the system architecture should be designed to be lightweight and scalable with minimal runtime overhead and decision complexity.

III. FLAME DESIGN

Fig. 4 provides a system overview of **`Flame`**, which employs a two-layer cache management design and operates as a back-end module in the FaaS platform. **`Flame`** comprises a global *CacheManager* and multiple *Cachelets* running on each worker. The global *CacheManager* communicates with each *Cachelet* and periodically gathers cluster-level information (e.g., worker status and workload behaviors) to make cache decisions. When a user request arrives, the front-end gateway dispatches it to a worker hosting the cached instance, if available. Otherwise, a cache miss triggers the cache-scheduling algorithm in the *CacheManager*, which selects a worker with the lowest hotspot contention to launch a new function instance. Upon completion

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.

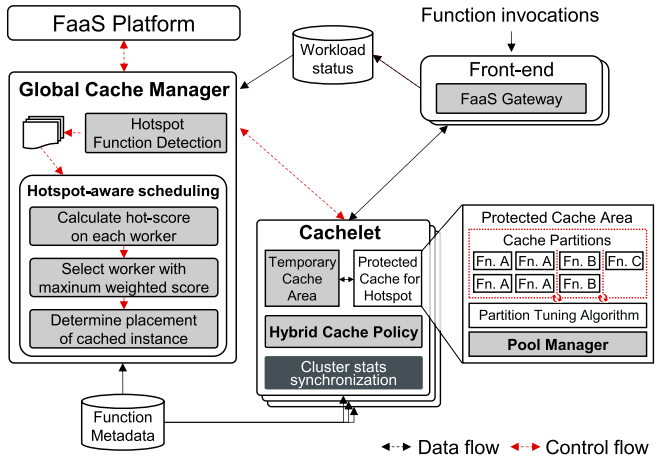


Fig. 4. The system overview of flame. It consists of a global *CacheManager* and multiple *Cachelets* running on each worker. We highlight the design of the partition-based cache pool and *Pool Manager*.

of function execution, the *Cachelet* caches the instance in memory and manages its lifetime following the *CacheManager*'s instructions.

When launching **Flame** in a FaaS platform, its initialization and operation involve two key steps: *function registration* and *cache management*. (1) *Function Registration*: For each function, **Flame** first gathers its metadata information and registers the function in a cluster-wide repository provided by the platform. The metadata includes the function ID, image name, and memory usage, which are utilized for cache allocation and reclamation in **Flame**. Function registration is performed only once for each new deployed function. (2) *Cache Management*: The cache management mechanism of **Flame** operates on two planes: the control plane and the data plane. The control plane, managed by the global *CacheManager*, determines whether a function should be cached and where it should be scheduled. Typically, hot functions are prioritized for caching over non-hot functions because they offer greater cache benefits. The data plane, consisting of the *Cachelet* and cache partitions on each worker, assigns hot functions to different partitions and dynamically adjusts their partition sizes.

Flame adopts a hotspot-aware cache scheduling with a partition-based cache pool design to address the "4H" questions. At the control plane, the *CacheManager* maintains a global hot function table and designs a *dynamic hotspot detection* mechanism to identify the cluster-level hot functions (§ IV-A). To avoid hotspot contentions due to workload skewness, it also introduces a "hotspot" concept for each worker, which is used to describe their load pressures. When a hot function instance needs to be created, the *CacheManager* calculates the hot-score on each worker and schedules a cached instance based on a *minimum hotspot aggregation* principle, which can balance both the server load and cache hit ratio (§ IV-B).

On each worker, **Flame** uses a hybrid cache policy to maximize cache efficiency. At the cache plane, the cache pool consists of two areas: *protected cache area* and *temporary cache area*. When a hot function instance is scheduled on a worker, it is cached in the protected area after invocation. The protected area is further divided into several partitions. A *Pool Manager* is responsible for mapping hot functions to different partitions and uses a partition-tuning algorithm to dynamically determine cache allocation for each hot function (akin to a priority-based cache policy). In comparison, the non-hot functions are cached in the temporary area in a best-effort manner (akin to a TTL-based cache policy, but can be evicted at any time if required) to utilize a worker's idle cache resources. The *Cachelet* periodically synchronizes the hot function table from *CacheManager* and makes cache decisions (§ IV-C).

The cache design within the worker node has several advantages: Firstly, the hybrid cache policy can prevent non-hot functions from becoming "second-class citizens" and suffering from extreme performance degradation, thus reducing the impact of hotspot detection delays or errors and improving system robustness. Second, the partition-based cache pool design enables more fine-grained cache allocation among local hot functions, thereby reducing intra-worker cache contention and improving performance fluctuations across tenants. Third, the partition tuning algorithm can promptly reclaim over-provisioned cache based on performance feedback from the hot function, and, together with the TTL-based cache policy for the non-hot function, eliminate cache redundancy to reduce cache cost.

IV. Key Components of Flame

We now introduce the three key modules of **Flame**: *dynamic hotspot detection, hotspot-aware instance scheduling*, and *cache resource allocation*, and explain how these mechanisms work together.

A. Dynamic Hotspot Detection

The *dynamic hotspot detection* mechanism is used to identify the global hot functions in the FaaS cluster. Inspired by Java Virtual Machine (JVM), the *CacheManager* uses an invocation counter to record the number of function invocations. Records collected from each worker are aggregated to derive cluster-level function hot-scores. Since most FaaS applications have varying workloads, the *CacheManager* dynamically records function invocations across several historical time intervals, each configurable (e.g., 1 hour). When the last interval ends, the function invocation counter is stored and reset to zero for the next time interval.

For each function, its hot-score is calculated from an *exponentially decaying* method, which can help **Flame** learn the historical workload and avoid hotspot detection oscillations caused by short-term workload bursts. Formally, the hot-score is defined as $H_i = \sum_{j=1}^{T} 2^{1-j} c_i[j]$, where $c_i(j)$ represents the jth invocation counter of function i and T is the total number of historical invocation counters. Given a collection of function hot-scores $\{H_i | i = 1, 2, ..., m\}$, where m is the number of functions. We sort the function hot-scores in descending order and define the top N functions as the hot functions, which meet the constraint $\sum_{i=1}^{N} H_i \geq r \cdot \sum_{i=1}^{m} H_i$, where r is a threshold named region size from 0 to 1. A larger (or smaller) region size means more (or fewer) of the most popular functions are

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.

classified as hot functions. As the region size setting can impact **Flame**'s cache decisions and resource efficiency, we conduct extensive experiments to determine the optimal region size and ultimately set it to 0.5 in practice (discussed in § VI-B).

B. Hotspot-Aware Instance Scheduling

The *hotspot-aware instance scheduling* mechanism determines the placement of newly launched function instances while alleviating hotspot contentions on local workers. The core principle of the scheduling algorithm is to *distribute function instances across the FaaS cluster according to the minimum hotspot aggregation principle*. To achieve this, we introduce a concept of "worker hot-score", which quantifies the load pressure on back-end servers. This score is the aggregated hotspot score for all hot functions cached on a specific worker. When a function instance requires scheduling, the *CacheManager* collects the status of all workers, computes their hot scores, and determines the optimal placement for the instance. It is important to note that the placement of function instances is equivalent to cache scheduling. Worker status is collected through a periodic synchronization mechanism between the *CacheManager* and *Cachelets*, occurring every 5 seconds, thereby minimizing communication overhead. Additionally, **Flame** supports forced synchronization to reduce scheduling failures.

Algorithm 1 shows the details. In a FaaS platform, a function's resource requirements can be represented as a multidimensional vector, encompassing CPU, memory, and I/O requirements. Let R denote the resource requirement of a newly launched function instance, and let $\boldsymbol{S}$ represent the available resources across all workers. Similarly, the hot-scores of functions and the resource consumption of deployed functions can be defined as $\boldsymbol{H}$ and $\boldsymbol{M}$, respectively. The scheduling algorithm takes these inputs and determines the target worker for launching the function instance.

During scheduling, the algorithm iterates over the workers and calculates their aggregated hot scores and available resources using the `getHotScore()` module (Lines 3-4). If the resource requirements of the newly launched instance can be satisfied, it will be scheduled on the worker with the lowest aggregated hot-score (Lines 5-8). Lines 10-27 provide further details of `getHotScore()`. The algorithm initializes F_k and I_k as lists of functions and instances on worker k, respectively, where k denotes the worker index. For each registered hot function on worker k, its hot-score is accumulated only if it has more than one cached instance on that worker (Lines 13-16); otherwise, it is skipped, since it does not compete for cache resources with other functions.

The available resources of a worker comprise unallocated resources and evictable resources in the temporary cache area. Typically, the available cache resources on each worker are predefined and limited (e.g., 40%). For each non-hot function on worker k, its cache resource usage is accumulated into the available resources of the worker since its cached instances can be forcibly evicted during the scheduling of new functions (Lines 17-21). As both insufficient worker resources and high hotspot workload pressure increase the risk of cache contention, we define a worker's scheduling priority as the ratio of its available resources to its aggregated hot-score (Lines 22-26). Workers with greater resource capacity and fewer hot functions are prioritized for scheduling new function instances, which helps alleviate workload skewness in the FaaS cluster.

Algorithm 1: Hotspot-aware Function Scheduling

Input:
R ▷ The resource requirement of the new launched function instance;
$\boldsymbol{S}$ ▷ The available resources of workers;
$\boldsymbol{H}$ ▷ The hot-score collection of the deployed functions;
$\boldsymbol{M}$ ▷ The resource consumption of the deployed functions;
Output:
$server\_index$ ▷ The placement of the new launched instance;

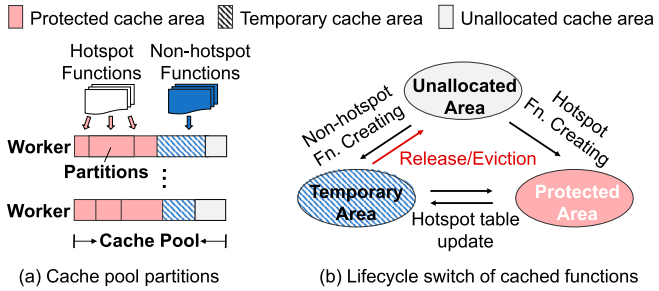
1 $max\_score \leftarrow 0$;
2 $server\_index \leftarrow -1$;
3 **for** $S_k \in \mathbf{S}$ **do**
4 $\quad \langle hot\_score_k, resource_k \rangle \leftarrow$ `getHotScore`$(S_k, \boldsymbol{H}, \boldsymbol{M})$; `// Calculate the aggregated function hot-score and the available resource on worker` k
5 $\quad$ **if** $R \leq resource_k$ **then**
6 $\quad\quad$ **if** $hot\_score_k > max\_score$ **then**
7 $\quad\quad\quad max\_score \leftarrow hot\_score_k$;
8 $\quad\quad\quad server\_index \leftarrow k$; `// Find a worker to schedule new instance`
9 **return** $server\_index$;
10 **Function** `getHotScore`$(S_k, \mathbf{H}, \mathbf{M})$**:**
11 $\quad hot\_score_k \leftarrow 0$; $resource_k \leftarrow 0$;
12 $\quad$ **Initialize** F_k, I_k as the cached functions and instances on worker k;
13 $\quad$ **for** $F_{k_i} \in F_k$ **do**
14 $\quad\quad$ **if** $I_{k_i} \neq []$ **then**
15 $\quad\quad\quad$ **if** $isHotspot(F_{k_i})$ **then**
16 $\quad\quad\quad\quad hot\_score_k \leftarrow hot\_score_k + H_i$;
17 $\quad\quad\quad$ **else**
18 $\quad\quad\quad\quad$ **for** $I_{k_i} \in I_k$ **do**
19 $\quad\quad\quad\quad\quad$ **if** $isIdle(I_{k_i})$ **then**
20 $\quad\quad\quad\quad\quad\quad resource_k \leftarrow resource_k + M_i$;
21 $\quad resource_k \leftarrow resource_k + S_k$; `// Update the worker's available resources`
22 $\quad$ **if** $resource_k > 0$ **then**
23 $\quad\quad$ **if** $hot\_score_k == 0$ **then**
24 $\quad\quad\quad hot\_score_k \leftarrow resource_k / 0.001$;
25 $\quad\quad$ **else**
26 $\quad\quad\quad hot\_score_k \leftarrow resource_k / hot\_score_k$; `// Calculate the weighted score`
27 $\quad$ **return** $\langle hot\_score_k, resource_k \rangle$;

Note that FaaS platforms typically use a separate cluster autoscaler (CA) to shut down or freeze underutilized workers. It is feasible to integrate **Flame** with CA to support workload consolidation when placing cached functions, thereby reducing the cost of the FaaS platform. Additionally, since caching function instances primarily consumes memory, the current scheduling algorithm considers only memory constraints. However, it can be easily extended to other resources such as the CPU and the network.

C. Cache Resource Allocation

As described in § III, the *Cachelet* employs a hybrid policy for caching hot functions and non-hot functions on each worker. Below, we elaborate on the design of the *Cachelet*, including the definitions of the protected and temporary cache areas, lifecycle

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.



(a) Cache pool partitions (b) Lifecycle switch of cached functions

Fig. 5. Cache allocation on each worker. Hot functions are cached in different partitions in a protected area, while non-hot functions are cached in a temporary area. The lifetime of cached functions are managed by *Cachelet*.

TABLE II
WORKLOADS USED IN FLAME'S EVALUATION

FaaS Workload	# of Func.	# of Requests	Avg. Reqs/s	[Top-N func.: >90% total invocs.] # of Functions	# of Mem Usage
Trace A	384	3,111,827	36	14 (3.64%)	40.57%
Trace B	384	11,363,701	132	4 (1.04%)	15.87%
Trace C	384	12,411,923	144	3 (0.78%)	69.09%
Trace D	384	24,774,632	287	2 (0.52%)	46.81%
Trace E	384	3,462,726	40	12 (3.12%)	68.77%
Trace F	384	2,735,329	32	16 (4.16%)	84.89%
Trace G	384	3,665,540	42	13 (3.38%)	12.35%
Trace H	384	2,183,501	25	17 (4.42%)	30.03%

management of cached functions within these areas, and the cache partition tuning for hot functions.

1) Protected & Temporary Cache Areas: For each worker, we assume that the FaaS platform predefines the maximum capacity of the cache pool and that all function instances, during both their execution and caching phases, operate within it. To implement the hybrid cache policy in *Cachelet*, we introduce two logical concepts: the *protected cache area* and the *temporary cache area*. As illustrated in Fig. 5(a), hot function instances are cached in the protected area, where they have no lifetime limitations and cannot be evicted due to resource contention (e.g., creating new function instances). If the cache pool has remaining unallocated space, it is utilized to cache non-hot functions. The cache usage for non-hot functions is considered temporary and can be automatically released or evicted when cache contention arises.

2) Lifecycle of Cached Functions: Fig. 5(b) illustrates the function cache lifecycle on each worker. Cache operations, such as cache allocation and area switching, can be triggered by several events, which are listed below:

Hot function Update: When the *CacheManager* updates the global hot function table, it notifies the *Cachelet* to synchronize and verify the cached instances on each worker. If a function is no longer classified as a hot function, all of its cached instances are reassigned to the temporary area. Conversely, if a function transitions to being a hot function, both its existing cached instances and newly launched instances are moved to the protected area.

Non-hot function Release/Eviction: For each non-hot function instance, the Cachelet employs a TTL-based keep-alive policy with a maximum keep-alive time limit. This limit is an empirically determined parameter derived from extensive experiments (discussed in § VI-B). Additionally, cached non-hot function instances may be evicted when there are insufficient cache resources to launch new function instances. This "best-effort" approach leverages unallocated cache resources to enhance function performance further.

3) Cache Pool Partitions: On each worker, the *Cachelet* employs a *Pool Manager* to manage the number of cache partitions and the functions mapped to them. For each hot function, its instances are cached in a dedicated partition, which isolates cache resources from one another and imposes a limit on the number of instances that can be cached. The *Pool Manager* monitors the function's average cache miss ratio and dynamically adjusts the corresponding partition size accordingly. In our testbed, the maximum number of partitions is set to 20, as we observe that only a small number of hot functions exist in our workload traces (see Table II). The partition tuning algorithm executes every 60 seconds and can handle most workload bursts encountered in FaaS scenarios.

The *Pool Manager* employs a heuristic algorithm to update the cache partitions, thereby determining appropriate cache resource allocation for each hot function based on performance feedback control. Algorithm 2 provides details. Initially, the *Pool Manager* uses a fair allocation strategy, assigning each cached function an equal-sized cache partition (Line 1). After initialization, the cache miss ratios of all cached functions are continuously monitored, and the cache partitions are adjusted according to each function's cache miss $slack$ value (Lines 2-16). The $slack$ is defined as follows

$$slack = (SLA\_target - cache\_miss)/cache\_miss,$$

where SLA_target represents the coldstart ratio SLA (Service Level Agreement) of a user function (e.g., $<1\%$ of coldstart ratio) and $cache\_miss$ is the average coldstart ratio during the last monitor time window. For function $l(s)$ with the largest (smallest) $slack$ (Lines 6-7), the *Pool Manager* updates their partitions following the operations below:

- If at least one function has little or negative $slack$, i.e., SLA is (about to be) violated, *Pool Manager* will assign 10% more resources to its cache partition, starting with function s with the smallest $slack$ by calling `upsize(s)` (Lines 8-10).
- When all functions comfortably satisfy their target SLAs, *Pool Manager* considers reducing 5% of resource allocation of function l that exhibits the highest cache miss $slack$. This allows excess resources to be reclaimed by `downsize(l)`, thus reducing the cache cost for the FaaS platform (Lines 12-14).

We use L and H as the decision thresholds (L, H $\in$ (0, 1)) for the above two cases, where $[H - L]$ represents the safety range of $slack$. Specifically, L (H) denotes the lower (upper) bound. A smaller L means a more aggressive cache allocation for each partition, while a larger H indicates a more conservative cache allocation behavior. In a production system, these thresholds can be configured based on the user's SLA requirements and the FaaS provider's cost budget. For example, given a 1% coldstart ratio SLA, if the cloud user is sensitive to

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2: Cache partition tuning.

```
1  Initialization(); // Beginning with an equal
     cache allocation for each partitions
2  while TURE do
3      update the worker cache pool size C;
4      update the cached hot function list F on the worker;
5      monitor average function cache miss in past 15 minutes;
6      find function s with the smallest slack;
7      find function l with the largest slack;
8      if slack[s] < L then
9          // At least one function may violate
             its SLA; prioritize the one with the
             worst performance
10         upsize(s); // Increase the partition
             size by 10%
11     else
12         if slack[l] > H then
13             // All functions have slack;
                 start reclaiming resources from
                 the one with the largest slack
14             downsize(l); // Reduce the partition
                 size by 5%
15     if cannot meet all functions' SLA targets for 30 minutes
         then
16         scaleupOrMigration(C,F); // Increase
             memory size by 10% for all
             partitions or migration when cache
             pool is exhausted
17 Function scaleupOrMigration(C, F):
18     calculate overall cache usage U of all partitions;
19     if U * 1.1 <= C then
20         // Check whether the cache partition
             usage exceeds the cache pool size
21         for f ∈ F do
22             upsize(f); // Increase the cache
                 pool size by 10%
23     else
24         stop launching new functions on this worker;
```

the function performance, a larger L can be set. Conversely, if the FaaS provider prefers to reduce cache cost, a smaller H can be configured to trigger the cache reclamation mechanism promptly. In our experiments, we set L and H to 0.05 and 0.2, respectively. The experimental results show it works well.

The *Pool Manager* also maintains a timer to monitor the duration of any ongoing SLA violations, which is reset once SLA requirements are satisfied. If no cache re-partition that meets all functions' SLA is found within 30 minutes, `scaleupOrMigration()` is triggered to increase the cache pool capacity or migrate functions to reduce server load if the cache pool size exceeds its capacity limit. The prolonged performance degradation can be prevented (Line 15-16). We also present the details of `scaleupOrMigration()` module (Lines 17-24).

Note that FaaS platforms may use preemptible cloud resources (such as harvest VM [13] to run workers, and the workers' available resources may change over time. We take this into account in Flame's system design. Algorithm 2 updates the worker cache pool size each time it runs, and checks whether the limitation of the worker cache pool is exceeded. If yes, the algorithm will trigger a migration operation, i.e., stopping scheduling or launching new function instances on this worker (Line 24). After that, the cache reclamation mechanism gradually reduces the partition's size and eventually converges on the cache pool's size limit.

V. Implementation

Flame is implemented on OpenFaaS [9], an event-driven FaaS platform built on top of Kubernetes, with approximately 5,000 lines of Golang code. We integrate a keep-alive policy into OpenFaaS to replace the default capacity-based scaling mechanism and introduce new components (e.g., a global *CacheManager* and *Cachelet*) for hot function detection and caching. This modification primarily involves the *gateway, faas-netes*, and *alert-manager* components. On each worker, the *Pool Manager* operates as a daemon thread within the *Cachelet* to manage cache partitions at the software level. Additionally, we develop 8,000 lines of code (in Java and Linux Shell scripts) for system simulation and testing.

Simulator: We design and implement a simulator to investigate the optimal system configuration for our caching policy and to validate its efficiency on a large-scale testbed. The Simulator is written in Java and consists of approximately 12,000 lines of code. It incorporates various caching policies to assist in determining the optimal parameter settings for **Flame**.

VI. Evaluation

A. Setup & Methodology

Testbed: We evaluate **Flame** on an 8-worker local cluster, where each worker is equipped with 256 GB of RAM and 32-core CPUs. All workers are interconnected via a 10 Gbps Ethernet network with full-bisection bandwidth. Additionally, we utilize the Simulator to determine the optimal parameter settings for **Flame**'s caching policy. The Simulator replicates the same workloads and evaluation environment as those in the real-world testbed. Large-scale experiments are also performed based on the simulation results.

Workload: We replay Azure's function traces [3] to generate the workload for our evaluation. These traces encompass function names, memory usage, request arrival timestamps, execution time, and startup time, collected from over 50,000 functions spanning two weeks. To align with our testbed, we utilize smaller subsets of these traces. Specifically, we categorize the functions into four groups based on their invocation counts and randomly select 96 functions from each group to construct a representative sample. As shown in Table II, we generate a total of 8 independent workload traces. Each trace comprises invocation records from 384 functions over 24 hours, exhibiting diverse workload behaviors.

Benchmarks: Since Azure's function traces do not include function code, we utilize benchmarks from ServerlessBench [10] and FunctionBench [11] to construct 384 real-world FaaS functions by configuring varying memory sizes and input parameters for the original benchmark applications. We deploy these functions and generate requests that follow the request arrival patterns specified in our eight workload traces. For each function in Azure's trace, we identify the closest match among our benchmarks based on both memory size and execution time similarity. The function's name, memory size, execution time, and startup time in these traces are then replaced with the corresponding actual values from our benchmarks.

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.

Comparison systems: We evaluate five different cache systems in our experiment and list them in the following:

- *OpenWhisk* [7]: Existing FaaS platforms, such as AWS Lambda and OpenWhisk, employ a TTL-based keep-alive policy to cache functions for a predefined duration (e.g., 15 minutes) after each invocation.
- *FaasCache* [5] (**ASPLOS '21**): FaasCache focuses on analyzing function behaviors, including popularity, memory consumption, and coldstart overhead, when caching functions in FaaS platforms. It adopts a priority-based greedy caching policy to determine which functions should remain cached on each worker.
- *CH-RLU* [8] (**HPDC '22**): CH-RLU aims to mitigate cache contention in FaaS clusters by balancing cache locality and worker load. It proposes an enhanced hashing-based algorithm with random forwarding to optimize workload distribution across the FaaS cluster.
- *Icebreaker* [14] (**ASPLOS '22**): Icebreaker addresses both function performance and keep-alive cost in heterogeneous FaaS clusters. It determines whether to cache a function on high-end machines, low-end machines, or not at all, based on factors such as coldstart ratio, resource over-provisioning, and server affinity. In our homogeneous testbed, we leverage Icebreaker's controller to decide whether to cache a function.
- *Flame-x*: We conduct an ablation study on **Flame** to evaluate the significance of each component design. In particular, we focus on evaluating the contributions of hotspot-aware function scheduling and a hybrid cache policy with a partition-based cache pool design. To assess their impact, we compare cache efficiency by separately (1) turning off hotspot-aware function scheduling (Flame-s), (2) turning off hotspot detection (Flame-h), and (3) disabling cache partitioning (Flame-p). The ablation evaluation results are presented in § VI-D.

Metrics: We consider three evaluation metrics: the coldstart ratio (i.e., the proportion of coldstarts invocations across all requests in the FaaS cluster), the function latency (i.e., the function performance impacted by function execution and coldstart events), and the overall cache cost (i.e., the memory consumption for keeping functions alive and executing requests).

B. Parameter Setting

Both the region size and keep-alive time setting can affect **Flame**'s cache efficiency. The former mainly determines the cache consumption in the protected area and the hit ratio for the hot function. The latter mainly affects the performance of the hotspot and non-hot functions. These two factors can also influence each other, making it, in fact, a complicated trade-off between cache cost and performance, especially for cache allocation across multiple tenants. To find the optimal parameter sittings, we measure the coldstart ratio and cache usage from extensive simulation experiments under different parameter combinations, where the keep-alive time changes from 1 minute to 15 minutes (every 1 minute) with three different region size settings (0.5, 0.7, and 0.9). The experimental results show that **Flame** achieves optimal performance with a region size of 0.5 and a 5-minute keep-alive time setting, which we call the "*Eden-setting*" and use in the following experiments.

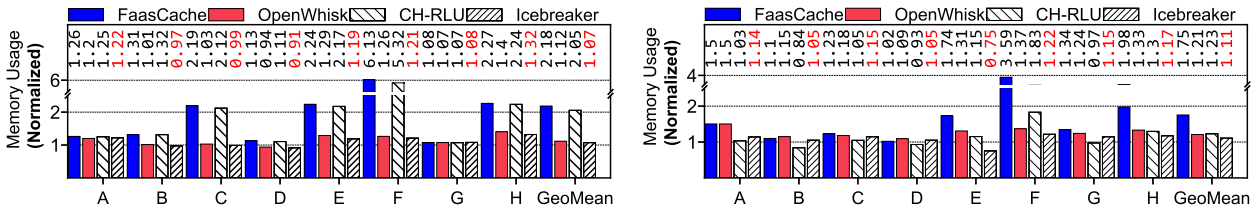
C. Overall Performance

Higher Efficiency: **Flame** achieves an average reduction in cache cost by 7%-56% and reduces the coldstart ratio by more than 7× compared to state-of-the-art methods. Figs. 6(a) and 7(a) illustrate the memory consumption and warm invocation ratio results under the hashing-based load-balancing rule, with **Flame** normalized to 1. The results demonstrate that **Flame** reduces overall memory consumption by 54% while maintaining a lower coldstart ratio compared to the state-of-the-art method, FaasCache. CH-RLU enhances FaasCache with a load-aware load balancer that randomizes request scheduling; however, this approach violates locality and results in a higher coldstart ratio than FaasCache. It is worth noting that although OpenWhisk consumes only half the cache resources as FaasCache, it incurs approximately 1.3× higher coldstart ratios. Icebreaker reduces overall cache usage by minimizing the number of cached functions based on workload prediction. However, the uncertainty in function access patterns across multiple workloads makes it challenging to make optimal decisions, resulting in approximately 3× higher coldstart ratios than **Flame**. During the evaluation of 8 workload traces, FaasCache, CH-RLU, OpenWhisk, and Icebreaker achieve warm invocation ratios of 97.1%, 87.2%, 89.3%, and 90%, respectively, relative to **Flame**.

Flame is designed to be generic and workload-agnostic in FaaS clusters, enabling it to adapt to various workload distributions. The evaluation results of **Flame** under the round-robin load-balancing rule are presented in Figs. 6(b) and 7(b). Unlike the hashing-based load-balancing rule, the round-robin rule distributes requests across multiple workers even when invocations originate from the same function. While this approach strictly balances workloads across workers, it violates cache locality. Consequently, all comparison systems require more cache resources (with an average increase of approximately 2× in memory usage) to maintain a low coldstart ratio. Despite this, **Flame** still achieves better performance. Compared to FaasCache, it reduces cache resource usage by approximately 42% (up to 72.1% in maximum cases) while maintaining a lower coldstart ratio. Although **Flame** reduces cache resource usage by only 18.7% and 9% compared to OpenWhisk and Icebreaker, respectively, the latter two methods exhibit higher coldstart ratios. Overall, **Flame** improves the coldstart ratio by nearly 3.5× and 3.3× on average when compared with OpenWhisk and Icebreaker, respectively.

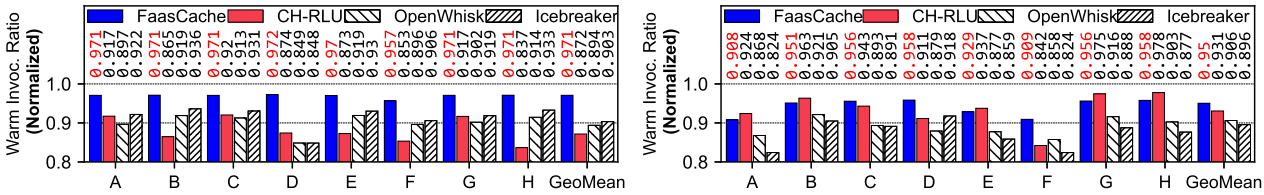
Flexible Cache Allocation: **Flame** dynamically identifies hot functions and adjusts cluster-level cache allocation based on workload changes. Its hybrid cache policy allocates resources for both hot and non-hot functions using a global cluster view. Fig. 8(a) shows the number of cached instances and memory usage under **Flame**'s control, demonstrating its ability to adapt cache decisions to workload variations. During bursts, **Flame**'s

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.



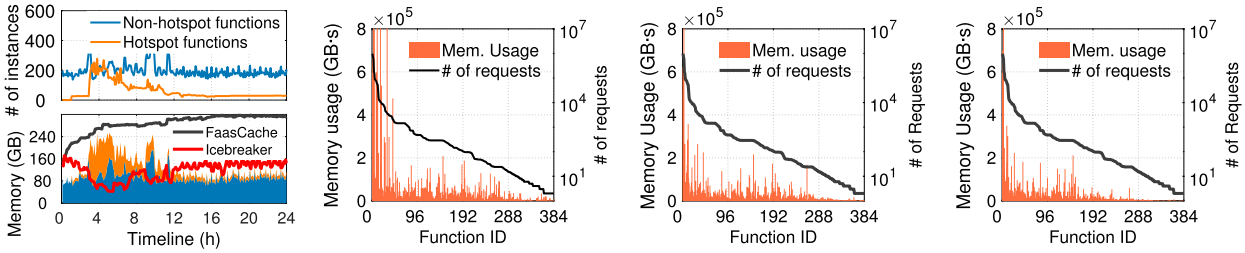
(a) Comparison under the hashing-based load-balancing (b) Comparison under the round-robin load-balancing

Fig. 6. Comparison of the memory usage under (a) hashing-based and (b) round-robin load-balancing rules. The results of flame are normalized to 1.



(a) Comparison under the hashing-based load-balancing (b) Comparison under the round-robin load-balancing

Fig. 7. Comparison of the warm invocation ratio under (a) hashing-based and (b) round-robin load-balancing rules. The results of flame are normalized to 1.



(a) Cache resource allocation (b) FaasCache (c) Icebreaker (d) Flame

Fig. 8. (a) The number of cached instances in flame's runtime process and the runtime cache resource usage in FaasCache, flame, and icebreaker; distribution of the function memory usage in (b) FaasCache, (c) icebreaker, and (d) flame. The number of invocations sorts the functions in descending order.

exponentially decaying algorithm detects hot functions and prevents oscillations in caching decisions. After the workload spike (between 2 h and 4 h), it reclaims over-provisioned instances to reduce costs. In contrast, FaasCache caches all used functions until resources are depleted, quickly exhausting the worker cache pool. Icebreaker predicts request arrivals but struggles with sudden workload spikes, leading to underestimated cache allocations and a high coldstart ratio despite lower resource consumption.

Less Redundancy: **Flame**'s hybrid cache policy and reclamation mechanism significantly reduce cache redundancy in FaaS clusters. To demonstrate the efficiency of cache decisions in **Flame**, we use trace *E* as an example to illustrate the memory consumption breakdown, which includes memory usage per function for caching instances and executing user requests. Fig. 8 presents the details, where functions are sorted by the number of invocations in descending order. In this example, FaasCache consumes 25,188 TB·s of memory to cache all functions due to its local cache control, which introduces significant cache redundancy across workers (Fig. 8(b)). Icebreaker does not distinguish hot functions but reduces overall function keep-alive time, resulting in only 13,463 TB·s of memory consumption. However, lower resource consumption does not necessarily imply a lower coldstart ratio (Fig. 8(c)). **Flame**'s hybrid cache policy considerably improves memory cost for most functions, consuming only approximately 11,800 TB·s of memory (Fig. 8(d)). Additionally, approximately 80% of memory savings come from the top-N functions that generate more than 90% of total requests, demonstrating the effectiveness of **Flame**'s hybrid cache policy and cache reclamation.

Fewer Hotspot Contentions: **Flame**'s hotspot-aware cache scheduling significantly reduces workload skewness and alleviates hotspot contentions in FaaS clusters. Workload skewness of hot functions can lead to cache resource contention, frequent scheduling failures, or request discards. Table III provides a breakdown of function coldstart ratios and request drop ratios for comparison systems. The drop ratio represents the

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.

TABLE III
BREAKDOWN OF COLDSTART RATIO AND DROP RATIO

Methods	Coldstart Ratio	Drop Ratio	Overall
Flame	0.67%	0.02%	0.69%
FaasCache	1.83% (↑2.7×)	0.77% (↑38.7×)	2.61% (↑3.8×)
CH-RLU	5.34% (↑7.9×)	1.41% (↑70.5×)	6.75% (↑9.8×)
OpenWhisk	4.68% (↑7.0×)	1.63% (↑81.4×)	6.31% (↑9.1×)
Icebreaker	6.11% (↑9.1×)	1.34% (↑66.7×)	7.45% (↑10.8×)

fraction of requests that cannot be processed due to unavailable function instances. We observe that **Flame**'s hotspot-aware cache scheduling greatly reduces cache contention, achieving only 0.67% coldstart ratios and 0.02% drop ratios. Among the remaining four systems, FaasCache has the smallest drop ratio (0.77%), as its priority-based caching policy evicts unpopular functions to free up cache resources. Compared to OpenWhisk (1.63% drop ratio), CH-RLU reduces resource contention (1.41% drop ratio) but exhibits a higher coldstart ratio (5.34%) due to lower cache reuse. Icebreaker's caching algorithm reduces the number of cached functions, improving the request drop ratio by approximately 20% compared to OpenWhisk. However, it also results in more coldstart invocations (6.11% coldstart ratio). In summary, **Flame**'s centralized caching decisions reduce hotspot contentions by 38.7×-81.4× compared to existing local cache control methods, achieving 3.8×-10.8× performance improvements over them.

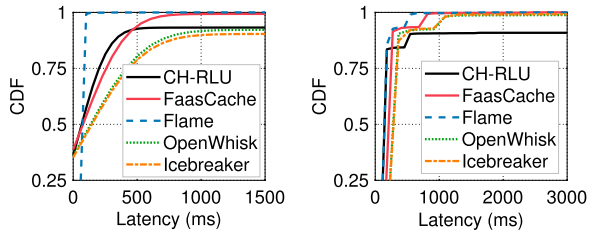
Lower Request Latency: **Flame** reduces the 99th percentile latency by more than 10× through mitigating coldstart overhead. Fig. 9 depicts the CDF of request latency for different comparison systems. For each system, we aggregate all function invocations under the eight workload traces to analyze their latency distributions. We observe that **Flame** achieves the lowest function latency under both hashing-based and round-robin load-balancing rules. CH-RLU demonstrates better function performance than OpenWhisk and Icebreaker under consolidated workload distributions (e.g., consistent hashing in Fig. 9(a)). However, its caching efficiency decreases when using a load-balanced request dispatcher (round-robin in Fig. 9(b)). FaasCache's priority-based caching policy assigns lower priority to functions with longer startup times, resulting in longer tail latency than CH-RLU. Nevertheless, its smaller coldstart ratio enables it to achieve lower function latency than OpenWhisk and Icebreaker. Icebreaker and OpenWhisk exhibit similar latency distributions because both adopt a fixed-duration caching policy, leading to high coldstart ratios, especially under unpredictable workload patterns. Consequently, they have the worst function performance (more than 10× increase in $P99$ latency compared to **Flame**) among the five comparison systems.

D. Ablation Study

In this section, we further evaluate **Flame**'s cache efficiency for each of its core component designs under four representative workloads (Trace A, C, F, and H; some similar workloads are omitted). We also present details of the system's runtime profiles to help illustrate the importance of these techniques.

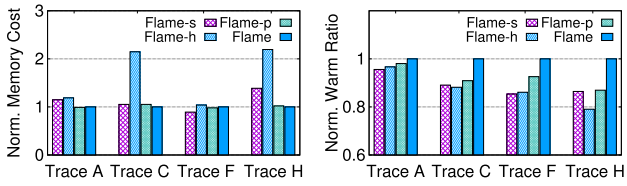
Component analysis: All of Flame's techniques contribute together to improve the function coldstart overhead and cache cost. We present the memory usage and warm invocation ratio results of the ablation study in Fig. 10(a) and 10(b), respectively. The results of complete **Flame** are normalized to 1. We can see that all **Flame**'s features have an impact on the overall performance, with Flame-h affecting cache usage most (disabling it results in an additional 2× of memory consumption), and both Flame-s and Flame-p impact the function coldstart ratios (disabling them results in a 5%-20% of maximum coldstart ratio reductions). Note that there is no noticeable change in memory usage since the cache partition is not designed for saving cache cost, but it enables **Flame** a better performance than the existing monolithic cache pool design on each worker. Overall, it can be observed that through the collaboration of different components or feature design, **Flame** can efficiently and effectively address the coldstart problem in FaaS platform.

Runtime profiles: Flame's partition tuning algorithm enables a fine-grained cache allocation among hot functions based on their SLA feedbacks and significantly reduces the performance fluctuations from cache contentions. Fig. 11(a) illustrates the worker-level runtime profiles of *Cachelet*'s cache control in **Flame**, where the profile of Flame-s is not shown since its poor performance. From the timeline curves, we can see that the cache partitioning technique achieves a relatively low coldstart ratio. Although there are many fluctuations in cache usage, we always combine the cluster-level hot-function scheduling with the worker-level cache partitioning mechanism in practice to avoid significant performance fluctuations. We also present the coldstart ratio distribution across different functions in Fig. 11(b), which shows that **Flame** with cache partitioning has a more concentrated distribution than the comparison systems. For the best case, **Flame** can reduce the coldstart ratio variation across different functions by 2×-4×. Even in the worst case, it still achieves approximately 50% of



(a) Hashing-based load-balancing (b) Round-robin load-balancing

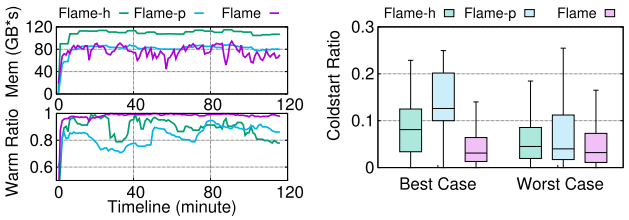
Fig. 9. Function latency distributions of five caching methods under the (a) hashing-based and (b) round-robin load-balancing rules.



(a) Overall memory cost (b) Warm invocation ratio

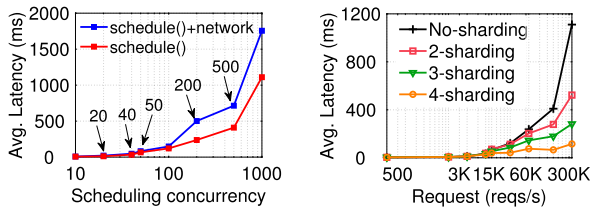
Fig. 10. Performance comparison of flame-s, flame-h, flame-p and origin flame under four representative workloads: (a) normalized overall memory cost; (b) overall warm invocation ratio, defined as '1 - coldstart ratio'.

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.



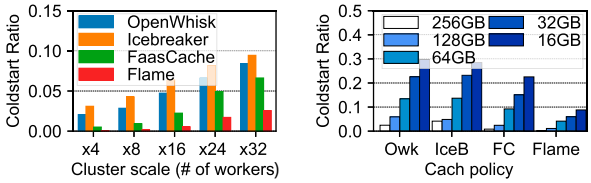
(a) Runtime profiles (b) Distribution of coldstart ratio

Fig. 11. More details in flame-h, flame-p and flame: (a) runtime profiles in memory usage and warm invocation ratio; (b) distributions of coldstart ratio.



(a) Cache scheduling time (b) Scheduling time optimization

Fig. 13. **Flame**'s scheduling overhead and optimization method.



(a) Scalability with cluster scales (b) Scalability with worker sizes

Fig. 12. Performance evaluation of flame by changing (a) cluster scale and (b) worker cache pool size. Owk: OpenWhisk; IceB: Icebreaker; FC: FaasCache.

improvement. This means that the cache partition policy can improve fairness and guarantee better performance in multi-tenant scenarios.

E. Discussion

System Scalability: **Flame** achieves high cache efficiency in large-scale clusters or clusters with limited server memory. To evaluate the cache efficiency of **Flame** under varying cluster scales, we adjust the number of servers in the testbed while maintaining a constant total memory capacity. Fig. 12(a) compares the coldstart ratio across cluster sizes of 4, 8, 16, 24, and 32 workers. We observe that the coldstart ratio in OpenWhisk, FaasCache, and Icebreaker increases significantly as the cluster scale expands. This is attributed to their local cache control mechanisms, which lead to increased hotspot contention and cache redundancy in smaller-sized workers. Consequently, the overall function performance in these methods deteriorates. In contrast, **Flame** maintains a consistently lower coldstart ratio. As shown in Fig. 12(b), we find that FaasCache, OpenWhisk, and Icebreaker all experience high coldstart ratios when reducing the worker memory. Although the coldstart ratio of **Flame** rises to nearly 8% when worker memory scales below 128 GB, it still achieves the lowest coldstart ratio.

System Overhead: **Flame** incurs minimal system overhead, enabling it to be easily scalable for large-scale practical deployments. We further analyze the system overhead of **Flame**'s centralized cache controller. **Flame** primarily consists of three sources of overhead:

(1) **Hot Function Detection (HFD):** HFD is a daemon process in *CacheManager*. It calculates the hot score in parallel for each function, ensuring minimal overhead regardless of the number of functions or cluster size. Our measurements show it takes 1.5 ms to process 384 functions.

(2) **Function Scheduling (FS):** FS is a key module in the *CacheManager* invoked only during cache misses. With a low cache-miss ratio, even high concurrent request volumes result in minimal FS invocations. The algorithm incurs 2 ms overhead on a 384-function testbed. Under 200 cache misses per second, FS generates 240 ms decision latency (Fig. 13(a)). Given a 1% coldstart ratio, **Flame** can theoretically handle up to 83K reqs/s ($max\_throughput = \frac{200}{0.24 \times 0.01}$).

(3) **Hybrid Cache Policy (HCP):** HCP is a daemon process in *Cachelet*. It monitors instance lifetimes in parallel. Updating the global hot-function list for each *Cachelet* incurs negligible overhead (~800 us). Each worker periodically runs two threads every 60 seconds: (i) updating cache areas and (ii) tuning partition sizes for hot functions. Invoking Kubernetes APIs to create or delete instances takes less than 1 ms.

Additionally, **Flame** adopts a lightweight design with minimal resource usage. The *CacheManager* consumes 0.03 CPU cores and 175 MB memory, while each *Cachelet* uses 0.05 CPU cores and 100 MB memory. In a FaaS cluster, **Flame** requires 2 CPU cores and 8 GB memory for the *CacheManager*, and all *Cachelet* consume approximately 25 CPU cores and 48 GB memory. This is relatively small for production environments.

In an extremely large-scale cluster, the centralized controller of **Flame** may face scalability challenges. To improve scalability, higher scheduling parallelism and workload sharding can be used. Evaluation results show that workload sharding significantly reduces decision-making pressure on the *CacheManager*. As shown in Fig. 13(b), at 300K reqs/s, the no-sharding design results in an average decision latency of over 1000 ms per function. With 3-sharding, this drops to approximately 200 ms, improving performance by more than 5×. We recommend adopting sharding to enhance **Flame**'s scalability in large-scale FaaS scenarios.

Economic Benefit: **Flame** can enable FaaS providers to reduce function cache costs by more than 40%. We conclude by discussing the potential cost savings for FaaS providers achieved through **Flame**'s optimizations. To evaluate the economic benefit, we measure each caching policy's total

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
COMPARISON OF CACHE COST (COLDSTART RATIO <1%)

	OpenWhisk	FaasCache	Flame	Icebreaker
Mem usage/req	1132 MB·s	663 MB·s	370 MB·s	988 MB·s
Cache cost/req [$]	1.85×10^{-5}	1.08×10^{-5}	6.08×10^{-6}	1.62×10^{-5}

memory consumption and calculate the average memory cost per invocation. Following the pricing model of AWS Lambda in the Eastern United States (Ohio) [15], we set the cost of 1 GB of memory to 0.06048 $/hour. Table IV demonstrates that **Flame** reduces the memory cost per request by approximately 44%, 67%, and 62% compared to FaasCache, OpenWhisk, and Icebreaker, respectively. Considering the scale of our production system, which deploys over 40,000 FaaS functions and serves 1.3 billion requests daily, the annual memory cost for caching functions is approximately 8,810,000$. By adopting **Flame** to replace the default cache policy, the annual memory cost decreases to 2,880,000$, resulting in an annual saving of approximately 5,900,000$.

Heterogeneous Deployment: The current version of **Flame** concentrates on optimizing function caching strategies in homogeneous clusters. Although it significantly reduces cache resource consumption (primarily memory) compared to existing approaches, FaaS providers may still face expensive costs inevitably. Several studies have explored caching strategies under heterogeneous cluster configurations [14], [16], [17], from which **Flame** can draw inspiration. For instance, cold functions can be offloaded to servers equipped with older CPUs and memory devices to reduce the cost budget, or functions that experience frequent coldstarts can be cached on high-performance worker nodes with faster instance startup speeds to achieve fairness among tenants. These insights will inspire our future research directions.

VII. RELATED WORK

Startup Acceleration: To mitigate the coldstart overhead, various techniques have been proposed to reduce function startup time. During the sandbox initialization phase, function instances can leverage lightweight virtualization technologies, such as launching sandboxes from trimmed containers with microVM solutions (e.g., FireCracker [18], Kata [19], SOCK [4]) or security containers (e.g., gVisor [20]). Additionally, methods like instance pooling [7], [21], function pre-warming [3], [22], [23], and environment sharing [24], [25], [26], [27] can effectively bypass the sandbox startup phase, offering promising approaches to reduce latency. However, these methods do not eliminate coldstarts, as the application startup overhead remains [1], [28]. While techniques such as code/library trimming [1] and pre-loading [28] can further reduce user code initialization time, their applicability is often limited to specific scenarios or programming languages. In contrast, **Flame** aims to avoid coldstart invocations by caching the entire execution environment, achieving near-zero coldstart latency and remaining orthogonal to these acceleration techniques.

Function Caching: Function caching represents another approach to prevent coldstarts. FaaS providers typically adopt static keep-alive policies or simple "warm container pools" [7], which are straightforward to implement but may incur high keep-alive costs. Prior works have explored learning-based [3], [14], [29] and priority-based caching policies [5] to improve cache efficiency; however, their effectiveness can be influenced by workload skewness and cache contention. Although some studies propose load-balancing techniques [8] to address these issues, intra-worker hotspot contention persists under monolithic cache pool designs. **Flame** can mitigate both inter-worker and intra-worker cache contention and achieves better performance without increasing cache costs.

Other Techniques: Furthermore, snapshot mechanisms for microVMs and containers, such as those based on CRIU [12], [30], [31], [32], [33], can significantly reduce sandbox startup time to tens of milliseconds. Launching functions from libOS environments [34] or WebAssembly (WASM) platforms [35] has also garnered significant attention, particularly for resource-constrained scenarios such as edge computing. When deploying functions across workers, modern hardware or protocols like RDMA [6] and CXL [36], [37] can help reduce coldstart overhead by accelerating or skipping function image transmission. While adopting these approaches may necessitate substantial modifications to cloud infrastructure or operating systems, **Flame** offers a more general solution and remains orthogonal to these techniques.

VIII. CONCLUSION

Up to now, Flame has been deployed and runs in a commercial production system for over 10 months. In this paper, we aim to present more details of Flame's design, including both the centralized cache controller and the new cache partitioning techniques added to each worker. We hope that our preliminary study will attract more attention and foster deeper discussions in this research area. In the future, we plan to investigate techniques such as memory tiering and hierarchical caching to enhance function performance and cache efficiency further.

ACKNOWLEDGMENT

We thank the anonymous reviewers and editors for their valuable feedback.

REFERENCES

[1] X. Liu et al., "FaaSLight: General application-level cold-start latency optimization for function-as-a-service in serverless computing," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, pp. 119:1–119:29, 2023.
[2] "Keeping functions warm." Amazon. Accessed: Dec., 2025. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html
[3] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, USENIX Association, 2020, pp. 205–218.
[4] E. Oakes et al., "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proc. USENIX Annu. Tech. Conf., (USENIX ATC)*, Boston, MA, USA. USENIX Association, 2018, pp. 57–70.
[5] A. Fuerst and P. Sharma, "FaasCache: Keeping serverless computing alive with greedy-dual caching," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, New York, NY, USA: ACM, 2021, pp. 386–400.

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.

[6] X. Wei et al., "No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing," in *Proc. 17th USENIX Symp. Oper. Syst. Des. Implementation (OSDI)*, Boston, MA, USA, USENIX Association, 2023, pp. 497–517.
[7] "Apache OpenWhisk: Open source serverless cloud platform." OpenWhisk. Accessed: May, 2016. [Online]. Available: https://openwhisk.apache.org/
[8] A. Fuerst and P. Sharma, "Locality-aware load-balancing for serverless clusters," in *Proc. 31st Int. Symp. High-Perform. Parallel Distrib. Comput. (HPDC)*, New York, NY, USA: ACM, 2022, pp. 227–239.
[9] OpenFaas. Accessed: Aug., 2024. [Online]. Available: https://docs.openfaas.com/
[10] T. Yu et al., "Characterizing serverless platforms with serverlessbench," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, New York, NY, USA: ACM, 2020, pp. 30–44.
[11] J. Kim and K. Lee, "FunctionBench: A suite of workloads for serverless cloud function service," in *Proc. 12th IEEE Int. Conf. Cloud Comput. (CLOUD)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 502–504.
[12] D. Du et al., "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Lausanne, Switzerland. New York, NY, USA: ACM, 2020, pp. 467–481.
[13] Y. Zhang et al., "Faster and cheaper serverless computing on harvested resources," in *Proc. ACM SIGOPS 28th Symp. Oper. Syst. Princ. (SOSP)*, Koblenz, Germany. New York, NY, USA: ACM, 2021, pp. 724–739.
[14] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *Proc.27th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Lausanne, Switzerland. New York, NY, USA: ACM, 2022, pp. 753–767.
[15] "AWS lambda price." Amazon. Accessed: Dec., 2025. [Online]. Available: https://aws.amazon.com/cn/lambda/pricing/
[16] R. B. Roy, T. Patel, R. Garg, and D. Tiwari, "CodeCrunch: Improving serverless performance via function compression and cost-aware warmup location optimization," in *Proc. 29th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, La Jolla, CA, USA, R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, Eds. New York, NY, USA: ACM, 2024, pp. 85–101.
[17] C. Lu, H. Xu, Y. Li, W. Chen, K. Ye, and C. Xu, "Smiless: Serving DAG-based inference with dynamic invocations under serverless computing," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage, Anal. (SC)*, Atlanta, GA, USA, Piscataway, NJ, USA: IEEE Press, 2024, p. 38.
[18] A. Agache et al., "Firecracker: Lightweight virtualization for serverless applications," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation (NSDI)*, Santa Clara, CA, USA. USENIX Association, 2020, pp. 419–434.
[19] A. Randazzo and I. Tinnirello, "Kata containers: An emerging architecture for enabling MEC services in fast and secure way," in *Proc. 6th Int. Conf. Internet Things, Syst., Manage. Secur., (IOTSMS)*, Granada, Spain, Piscataway, NJ, USA: IEEE Press, 2019, pp. 209–214.
[20] "GVisor: Application kernel for containers." GVisor. Accessed: Jun., 2025. [Online]. Available: https://gvisor.dev/
[21] Q. Liu et al., "Harmonizing efficiency and practicability: Optimizing resource utilization in serverless computing with Jiagu," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Santa Clara, CA, USA, USENIX Association, 2024, pp. 1–17.
[22] Z. Zhou, Y. Zhang, and C. Delimitrou, "AQUATOPE: QoS-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proc. 28th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Vancouver, BC, Canada. New York, NY, USA: ACM, 2023, pp. 1–14.
[23] R. B. Roy, T. Patel, and D. Tiwari, "Daydream: Executing dynamic scientific workflows on serverless platforms with hot starts," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Dallas, TX, USA, Piscataway, NJ, USA: IEEE Press, 2022, pp. 22:1–22:18.
[24] I. E. Akkus et al., "SAND: Towards high-performance serverless computing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Boston, MA, USA. USENIX Association, 2018, pp. 923–935.
[25] D. Saxena, T. Ji, A. Singhvi, J. Khalid, and A. Akella, "Memory deduplication for serverless computing with Medes," in *Proc. 17th Eur. Conf. Comput. Syst.*, Rennes, France, New York, NY, USA: ACM, 2022, pp. 714–729.
[26] Z. Li et al., "Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Carlsbad, CA, USA. USENIX Association, 2022, pp. 69–84.
[27] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, USENIX Association, 2021, pp. 805–820.
[28] Y. Sui, H. Yu, Y. Hu, J. Li, and H. Wang, "Pre-warming is not enough: Accelerating serverless inference with opportunistic pre-loading," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, Redmond, WA, USA. New York, NY, USA: ACM, 2024, pp. 178–195.
[29] M. Abdi et al., "Palette load balancing: Locality hints for serverless functions," in *Proc. 18th Eur. Conf. Comput. Syst. (EuroSys)*, Rome, Italy. New York, NY, USA: ACM, 2023, pp. 365–380.
[30] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "SEUSS: Skip redundant paths to make serverless fast," in *Proc. 15th EuroSys Conf. (EuroSys)*, Heraklion, Greece, New York, NY, USA: ACM, 2020, pp. 32:1–32:15.
[31] L. Ao, G. Porter, and G. M. Voelker, "FaaSnap: FaaS made fast using snapshot-based VMs," in *Proc. 17th Eur. Conf. Comput. Syst. (EuroSys)*, Rennes, France. New York, NY, USA: ACM, 2022, pp. 730–746.
[32] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, New York, NY, USA: ACM, 2021, pp. 559–572.
[33] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proc. 21st Int. Middleware Conf. (Middleware)*, Delft, The Netherlands, New York, NY, USA: ACM, 2020, pp. 1–13.
[34] A. Madhavapeddy et al., "Unikernels: Library operating systems for the cloud," in *Proc. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Houston, TX, USA. New York, NY, USA: ACM, 2013, pp. 461–472.
[35] J. You et al., "AlloyStack: A library operating system for serverless workflow applications," in *Proc. 20th Eur. Conf. Comput. Syst. (EuroSys)*, Rotterdam, The Netherlands. New York, NY, USA: ACM, 2025, pp. 921–937.
[36] J. Huang et al., "TrEnv: Transparently share serverless execution environments across different functions and nodes," in *Proc. ACM SIGOPS 30th Symp. Oper. Syst. Princ. (SOSP)*, New York, NY, USA: ACM, 2024, pp. 421–437.
[37] C. Alverti, S. Psomadakis, B. Ocalan, S. Jaiswal, T. Xu, and J. Torrellas, "CXLfork: Fast remote fork over CXL fabrics," in *Proc. 30th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Rotterdam, The Netherlands. New York, NY, USA: ACM, 2025, pp. 210–226.

Yanan Yang received the Ph.D. degree from the College of Intelligence and Computing, Tianjin University, China, in 2024. He is currently a Researcher with China Telecom Corporation Ltd. His research interests include cloud computing and serverless computing. He has published in Eurosys, SC, ASPLOS, ATC, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, and *ACM Transactions on Computer Systems*, and received the ACM SIGOPS Outstanding Doctoral Dissertation Award in 2024.

Wenda Tang received the B.S. and M.S. degrees from Xidian University, China, in 2013 and 2016, and the Ph.D. degree from Nanjing University, China, in 2019, majoring in computer science and technology. He is currently a Principal Researcher with China Telecom Cloud Computing Research Institute. His research interests include parallel and distributed computing, cloud computing, serverless computing, and disaggregated datacenter architectures.

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.

Laiping Zhao received the Ph.D. degree from the Department of Informatics, Kyushu University, Japan, in 2012. He is currently a Full Professor with the College of Intelligence and Computing, Tianjin University, serving as an Associate Dean with the School of Software. His research topics interests include Cloud Computing and LLM Systems. He has published at prestigious conferences such as SC, ASPLOS, ATC, and EuroSys, as well as in top-tier journals such as IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON COMPUTERS, and *ACM Transactions on Computer Systems*. He received the Best Paper Award at IFIP NPC'24 and ICPADS'22, the Best Paper Finalist at SC'21, and the Best Student Paper Award at AINA'12.

Keqiu Li (Fellow, IEEE) received the B.S. and M.S. degrees from the Department of Applied Mathematics, Dalian University of Technology, in 1994 and 1997, respectively, and the Ph.D. degree from the Graduate School of Information Science, Japan Advanced Institute of Science and Technology, in 2005. He is currently a Full Professor and Dean of the College of Intelligence and Computing with Tianjin University, China. He was a recipient of the National Science Foundation's Distinguished Young Scholars of China award. He is working on topics in blockchain, mobile computing, data centers, and cloud computing. He has published more than 150 papers in prestigious journals and conferences, including IEEE TRANSACTIONS ON NETWORKING, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON MOBILE COMPUTING, SIGCOMM, NSDI, ASPLOS, EuroSys, and ICNP.

Jie Wu (Fellow, IEEE) is the Laura H. Carnell Professor with Temple University and the Director of the Center for Networked Computing (CNC). His research interests include cloud computing and networks. He routtenberg-3646236serves on several editorial boards, including IEEE/ACM ToN. He is/was general Chair/Co-Chair for IEEE IPDPS'23, ACM MobiHoc'23, and IEEE CCGrid'24, as well as Program Chair/Co-Chair for IEEE INFOCOM'11 and CCF CNCC'13. He was an IEEE Computer Society Distinguished Visitor, ACM Distinguished Speaker, and Chair for the IEEE Technical Committee on Distributed Processing (TCDP). He is a member of the Academia Europaea (MAE). He is currently on leave, working at China Telecom as a Scientist in Cloud Computing.

Authorized licensed use limited to: Tsinghua University. Downloaded on March 24,2026 at 05:59:04 UTC from IEEE Xplore. Restrictions apply.