Genie: A Lightweight Serverless Infrastructure for In-Memory Key-Value Caching With Fine-Grained and Prompt Elasticity

Huijuan Xiao, Shixi Yang, Kai Zhang, Yinan Jing, *Member, IEEE*, Zhenying He, and X. Sean Wang, *Senior Member, IEEE*

***Abstract*—An increasing number of web applications require cloud in-memory key-value stores to minimize latency and achieve higher throughput. They generally have diverse characteristics and constantly changing traffic volumes, which require different computational and memory resources. A serverless in-memory key-value store characterized by elastic resource allocation and pay-as-you-go billing could satisfy the requirements of diverse and dynamic workloads. However, we find current serverless IMKVs fail to achieve fine-grained and prompt resource elasticity due to the limitations of their infrastructures. This paper proposes Genie, a lightweight serverless infrastructure for in-memory key-value caching with fine-grained and immediate elasticity. In Genie, a novel approach is adopted to enable dynamic and independent resource allocation to multiple tenants. It processes all arrived requests and estimates the vCPU consumption with a lightweight machine-learning approach for fine-grained billing. Moreover, Genie estimates the working set and dynamically resizes the allocated memory for hit ratio requirements. Evaluation results show that CPU estimation could be achieved every 100 microseconds without impacting system performance, and memory capacity could be adjusted by megabytes within seconds. The holistic design incurs 1% -2% performance degradation compared to our baseline. Moreover, Genie achieves an average of 58.3% CPU and 49.9% memory savings compared to AsparaDB for Memcache.**

***Index Terms*—In-memory key-value store, fine-grained resource elasticity, serverless infrastructure.**

I. Introduction

With the development of cloud computing, cloud in-memory key-value stores (IMKVs) are widely used to reduce latency and achieve high throughput for increasing web applications [1], [2]. They require both memory and CPU resources: memory for caching working sets of web applications, and CPU for request processing. Conventional cloud IMKVs are typically deployed on virtual machines with coarse-grained and tightly coupled CPU and memory resources [3], [4], [5]. A more expensive instance is generally allocated with more vCPUs and larger memory space. For instance, a `r5.large` node in Amazon DynamoDB Accelerator (DAX) [6] has 2 vCPUs and 16 GiB memory, and a `r5.xlarge` node has 4 vCPUs and 32 GiB memory. With such a coarse-grained resource allocation, an application that demands 3 vCPUs and 10 GiB memory has to select a `r5.xlarge` node, incurring 1 vCPU and 22 GiB memory reserved but unused. Moreover, the demands on CPU and memory for an IMKV instance are determined by distinct factors: computational resources are strongly associated with the IMKV request rate, while memory depends on the application's working set size [7], [8], [9]. Web applications also exhibit diverse workload characteristics and fluctuating traffic patterns, further complicating resource allocation. For example, Twitter workloads demonstrate varying working set sizes [10], Wikipedia access traces report memory demands changing by up to 2× [11], and Facebook workloads exhibit diurnal patterns or unpredictable traffic [12]. Overall, as web applications have diverse workload characteristics and constantly changing traffic volumes [10], [12], [13], the resource allocation of conventional cloud IMKVs often leads to significant resource underutilization.

Serverless is an emerging paradigm of cloud computing that frees users from managing underlying components, adjusts resources according to users' demands, and optimizes billing from a fixed strategy to a pay-as-you-go basis. A serverless in-memory key-value store, featuring elastic resource allocation and pay-as-you-go billing characteristics [14], is specifically designed to serve diverse and dynamic workloads in the cloud. It is expected to provide automatic and agile resource allocation in real-time and bill according to actual computational and memory usage. Current serverless IMKVs are implementations on virtual machines [15] and serverless functions [16]. However, they fail to achieve fine-grained and prompt resource elasticity due to their underlying infrastructures.

VM-based serverless IMKVs fail to achieve fine-grained and prompt resource elasticity. Recently, AWS released ElastiCache Serverless [15], a serverless in-memory key-value store that adopts virtual machines as the underlying component. ElastiCache Serverless automatically scales the VM cluster for resource elasticity and bills users based on their resource usage [17]. However, as we illustrated above, the resource inefficiency from VM-based allocation is still unsolved. Moreover, the auto-scaling has to take several minutes for migrating and rebalancing data between different VMs, which incurs delayed

Received 19 April 2024; revised 16 March 2025; accepted 22 March 2025. Date of publication 15 May 2025; date of current version 28 May 2025. This work was supported in part by the Project of Key R&D Program of Shandong Province under Grant 2024CXGC010113. Recommended for acceptance by S. Salihoglu.

The authors are with the School of Computer Science, Fudan University, Shanghai 200437, China (e-mail: zhangk@fudan.edu.cn).

Digital Object Identifier 10.1109/TKDE.2025.3556427

1041-4347 © 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: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

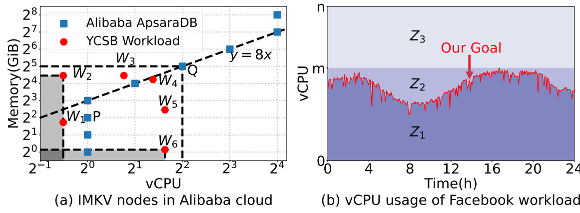

Fig. 1. Resources are tightly coupled to allocate with a coarse granularity.

resource elasticity [18] and non-negligible business loss on web applications [19]. Some studies have exploited serverless functions [16], [20] to construct serverless caching. However, since serverless functions are built on virtualized instances [3], [21] with short lifetimes and significant invocation times [16], [22], it is inefficient to implement high-performance in-memory caching atop them. Overall, current serverless IMKVs lack fine-grained and prompt elasticity owing to their underlying infrastructures.

In this paper, we propose Genie,[1] a lightweight infrastructure for serverless in-memory key-value caching with fine-grained and immediate elasticity. In Genie, a novel approach is employed to enable multiple tenants to efficiently utilize computational and memory resources, ensuring that these resources are dynamically and independently allocated. Computational resources are shared among tenants through processing requests from different tenants and are measured independently at millisecond granularity. Memory resources are managed through a shared pool and allocated to each tenant independently and dynamically. For computational resources, we find out critical factors of users' requests that affect computational usage. We train and implement a machine learning model to quantify the vCPU consumption according to users' workload characteristics and request rates. For memory resources, we design a flexible memory allocation mechanism to dynamically resize the memory capacity according to users' hit ratio requirements. Furthermore, we analyze the cause of inefficiency and propose adaptive caching to improve memory efficiency. Genie constantly monitors the memory, CPU, and network resource utilization and scales instantly to accommodate changes to the access patterns of each tenant's workload. As a lightweight infrastructure, it could be deployed on various underlying environments with fine-grained, immediate, and on-demand serverless caching.

In summary, the contributions are as follows:

- We propose a serverless IMKV infrastructure that achieves fine-grained and prompt elasticity in CPU and memory resources, respectively.
- We study workload characteristics that affect CPU usage and propose a lightweight machine-learning approach to estimate the vCPU usage of each tenant.
- We propose a flexible memory allocation mechanism to resize the memory capacity for each tenant to meet the hit ratio requirement and an adaptive mechanism that adjusts the eviction policy based on the workload distribution to improve memory efficiency.
- We design and implement Genie, a serverless infrastructure for in-memory key-value caching that achieves fine-grained and on-demand resource elasticity under diverse and dynamic workloads with minimal overhead.

[1] Genie is a magical spirit, originally in Arab traditional stories, who provides whatever the person who controls it asks.

II. Background and Motivation

A. Issues of Conventional Cloud Key-Value Stores

Nowadays, cloud vendors have provided various IMKVs to meet customers' needs, i.e., ElastiCache [23], Dynamo Accelerator (DAX) [5], ApsaraDB [24], etc. Specifically, they provide various IMKV nodes with different resource quotas, including a specific number of vCPUs, a specific amount of memory, and a certain network performance (e.g., a `r5.large` node in DAX is allocated with 2 vCPUs and 16 GiB memory [5]). These nodes derive from various virtualization approaches, including virtual machines (VMs) and containers, as well as disaggregated memory (DM) architectures that overcome the resource limitations of monolithic servers [3], [4], [25], [26], [27]. However, as plenty of applications with diverse characteristics and constantly changing traffic volumes emerge on the cloud [1], [28], we found conventional cloud IMKVs result in varying degrees of resource underutilization and inflexibility.

1) Resource underutilization when facing diverse workloads: Fig. 1(a) illustrates the resource underutilization of conventional cloud IMKVs [5], [23], [24] under diverse workloads. W_1 to W_6 are synthetic workloads that have the same characteristics as the well-known Yahoo! Cloud Serving Benchmark (YCSB) [29], but with different working sets and request rates. In ApsaraDB, W_1 necessitates a VM instance of type `P` to meet resource demands, while W_2, W_3, W_4, W_5, and W_6 all require instances of type `Q` due to the coarse-grained VM instances. From Fig. 1, we observe that the computational and memory resources of IMKVs' VM instances are coarse-grained with exponential growth and tightly coupled to allocate. In Fig. 1(a), both the memory capacities and the number of vCPUs grow exponentially, and a linear relationship exists between memory and compute resources.

However, for IMKV workloads, the demanded computational resources are strongly associated with the request rate, and the memory depends on the working set of the application [8], indicating that this tightly coupled approach does not reflect real-world requirements. For instance, the `USR` workload in Facebook exhibits high traffic while occupying only a small amount of memory [12], for which it requires a VM instance with a large number of vCPUs and a small amount of memory. Nevertheless, it has to select a VM instance in current cloud IMKVs that meets the vCPU requirement but memory capacity much greater than the working set size. It incurs a significant underutilization of memory resources. In Fig. 1, W_2 leads to an 82.5% under-utilized vCPUs and for W_6, the memory underutilization rate is 96.6%. This issue has been further highlighted by industry studies [30].

2) Inflexibility and underutilization under dynamic workloads: Research on key-value workloads reveals that different applications exhibit dynamic characteristics, leading to variable demands for resources over time [1], [10], [13]. Specifically,

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

the dynamic characteristics comprise request rates (e.g., traffic spikes and troughs of ETC workload in Facebook [12]), item sizes (e.g., sudden changes and periodic shifts in size distribution in Twitter [10]), the ratio of different operations [31], the skew factor (e.g., the changing skewness of Alibaba's production workloads [32]), and so on.

Fig. 1(b) depicts the vCPU utilization throughout a day from ETC workload in Facebook [12]. It presents an evident diurnal pattern characterized by alternating traffic spikes and troughs. m represents the maximum vCPUs required from the traffic spike, and n is the vCPU capacity from the most suitable VM. Zone Z_1 is the actual usage of computational resources, Z_2 represents the underutilization coming from users selecting an instance that could handle all of the traffic spikes, and Z_3 shows the vCPU underutilization from the coarse-grained provisioned instance.

When the computational resources required for peak traffic of ETC workload match the number of vCPUs in a single instance, the vCPU underutilization is minimized. Computational resources experience a 19.3% underutilization that comes from Z_2 (Z_3=0). However, when the vCPU required by the traffic peak slightly exceeds the computational capacity of a single VM, the coarse-grained resource allocation results in significant underutilization. In this scenario, users have to purchase a VM instance with twice the number of vCPUs to accommodate the traffic spike, resulting in an approximate underutilization of 59.7% (50% from Z_3 and 9.7% from Z_2).

Overall, the coarse-grained and tightly coupled resource allocation of conventional cloud IMKVs ignores the diverse and time-varying resource demands. It will easily result in resource underutilization and inflexibility.

B. Limitations of State-of-the-Art Serverless IMKVs

Serverless [14], [33] is an emerging paradigm of cloud computing that benefits users to focus on their own functionality of applications. It aims to relieve users from managing underlying components, auto-scale the services according to users' demand, and optimize the billing strategy to the pay-as-you-go model. A serverless in-memory key-value store characterized by *elastic resource allocation* and *pay-as-you-go billing* is an important implementation of serverless computing.

Specifically, *elastic resource allocation* refers to automatically adjusting the allocated computational resources based on the request traffic and resizing the memory capacity according to the working set size of the current workload [7], [8], [9]. Fine-grained and immediate resource elasticity further optimizes cost efficiency for tenants and enhances resource utilization for cloud providers, creating a win-win scenario that benefits both parties. Meanwhile, *pay-as-you-go billing* ensures users pay only for the CPU and memory resources their workloads actually occupy, making it a cost-effective and resource-efficient caching solution in the cloud. However, we find that state-of-the-art serverless IMKVs fail to achieve fine-grained and immediate resource elasticity due to their underlying infrastructures.

1) Serverless IMKVs based on VMs: Recently, AWS proposed a serverless in-memory key-value store, ElastiCache Serverless [15] that uses VMs as underlying components. It auto-scales the cluster of VMs to achieve resource elasticity. Specifically, the elasticity comes from horizontal scaling to add or remove VM nodes and vertical scaling to modify the resource capacities of VM nodes [17]. Compared to conventional cloud KVs [5], [23], [24], it relieves users from managing key-value instances and changes the billing to a usage-based model.

However, the frequent scaling for resource adjustment introduces significant overheads. Specifically, data migration and synchronization across different nodes consume additional CPU and network resources, as highlighted in previous studies [34], [35]. Moreover, the performance improvements expected from scaling up, or the resource savings anticipated from scaling down, are often delayed by minutes. This delay results from the time required for data migration and rebalancing [18]. Although the impact varies according to the data volume, the node type, and the ongoing traffic, the delayed elasticity will result in a non-negligible business loss on web applications [19]. Not only that, as described in Section II, the resource granularity it supports to allocate and resize is coarse-grained, which to some extent deviates from pay-as-you-go billing [15].

2) IMKVs based on serverless functions: Serverless functions are popular implementations of serverless computing [14], [22]. A serverless function is short-lived and stateless; they are provided by serverless infrastructures like AWS Lambda. Recent research has exploited serverless functions as underlying components to build serverless key-value stores [16], [20].

Nevertheless, research reveals that serverless functions are based on virtualized instances where vCPUs and memory are tightly coupled to allocate [3], [21]. For instance, in AWS Lambda [22], for every 1,769 MiB of memory allocated, one vCPU will be allocated to this function. This tightly coupled architecture results in an inability to scale one kind of resource independently, which limits the elasticity. Although there has been research on resource allocation for serverless functions [36], [37], the short lifetime of serverless functions (up to 15 minutes in AWS Lambda) and the significant invocation time (8 ms in AWS Lambda) make it inefficient to implement high-performance in-memory caching on serverless functions [16]. The evaluation in InfiniCache shows that it's not cost-effective when the request rate exceeds 86 requests per second [16]. In other words, the overhead and prices are significant when serving high-performance web applications.

Overall, state-of-the-art serverless IMKVs fall short of achieving fine-grained and prompt resource elasticity due to their underlying infrastructures.

III. Genie Design

In this section, we first clarify the goals and architecture of Genie. After that, we analyze the difficulties and challenges. Finally, we give an overview of the entire system.

A. Design Goal

We aim to propose a lightweight serverless infrastructure that implements serverless in-memory caching with fine-grained and immediate resource allocation. Our design will address the issues of coarse granularity and scaling delay in resource

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

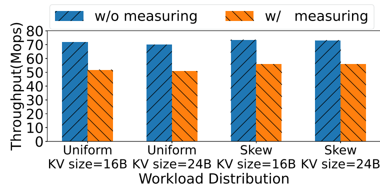

Fig. 2. Performance loss of measuring CPU time of each request.

adjustment in existing designs, thereby providing serverless caching for web applications in an agile manner.

Specifically, we have the following goals:

- Allocate the CPU and memory resources separately.
- Process as many requests as the user sends and measure the CPU consumption at the millisecond level.
- Dynamically resize the memory capacity to meet hit ratio requirements at MegaByte level and improve the memory efficiency of key-value items.

It has been reported that most tenants occupied very few resources in the cloud and approximately followed the heavy-tailed distribution [38]. The heavy-tailed distribution indicates that it's reasonable to simultaneously support dozens or even hundreds of tenants within an in-memory key-value caching with sufficient resources.

To enhance resource efficiency and reduce costs for cloud users, the serverless infrastructure we are going to propose is expected to share computational and memory resources with multiple tenants rather than using the conventional VM-based approach for isolation and allocation. The computational resources will be shared among different tenants through mixed traffic, and the memory resources will be managed separately and then allocated to each tenant to meet their hit ratio requirements. The serverless infrastructure aims to provide lightweight, fine-grained, and prompt resource elasticity for each tenant, which poses challenges to our design.

B. Challenge

The challenges for the fine-grained and immediate resource elasticity of Genie are as follows.

1) Measure the CPU consumption with small overhead: To get the computational resource consumption of each tenant in mixed traffic, directly measuring the time of processing each request on physical cores seems to be a straightforward solution. However, this approach is costly and inefficient for serverless in-memory key-value caching concerning tens of millions of key-value operations per second of the state-of-the-art key-value stores [39]. Fig. 2 reveals the performance degradation from measuring the CPU time of each request. Even the lightweight RDTSC is adopted for time measurement, it still results in a performance degradation of around 25%. Therefore, measuring each tenant's CPU consumption in a lightweight approach is a crucial challenge in Genie.

2) Efficiently resize the memory capacity: A slab-based or log-based item store is commonly used in key-value stores [8], [39], [40], where a slab divides memory into classes and smaller chunks for storing items, and a log stores items sequentially. However, a slab-based store leads to internal fragmentation and calcification, causing overhead and inflexibility in resizing memory [1], [8]. In contrast, a log-based approach provides fast insertions, easier defragmentation, and flexible memory reassignment without the need for reallocating memory among classes [39].

To resize memory in a log for hit ratio guarantees, a miss ratio curve (MRC) is used to illustrate the cache size and miss ratio relationship. However, a log exhibits various inefficiencies due to inherent characteristics, for which it's ineffective to resize the log based on the MRC directly. Besides, memory reclaim may lead to hit ratio fluctuations as all items on the outgoing memory will be evicted. The memory inefficiencies and hit ratio fluctuations pose challenges to memory elasticity.

C. System Overview

1) Our Approach: To achieve the goals we proposed in Section III-A, we design a serverless infrastructure Genie, that dynamically estimates the CPU usage and allocates memory for each tenant. Specifically, Genie manages compute and memory resources separately. It enables the whole computational resources to be shared among multiple tenants while dynamically resizing the memory of each tenant. It processes all of the tenants' requests in mixed traffic, and the resource utilization of each tenant is measured and estimated in real time to bill them in a fine-grained approach.

To estimate the CPU consumption with trivial overhead, we study characteristics that affect the CPU usage from users' workloads and quantify this impact through a learned approach. Genie generates workloads with diverse characteristics and gets the maximum throughput with different CPU cores. After that, it estimates the impact of workload characteristics on computational resource usage through a machine-learning model. While processing requests, Genie samples the requests of each tenant and calculates corresponding characteristics in real time. After that, it employs the machine learning model to measure the CPU usage of each tenant with a fine granularity.

To resize the memory capacity of each tenant with fine-grained while maintaining high performance, Genie adopts the append-only log as the underlying data structure to store key-value items. It proposes a flexible memory allocation mechanism to allocate and reclaim memory with minimal side effects. To determine the amount of memory each tenant should allocate, Genie periodically builds a miss ratio curve to model the cache locality. Then, it dynamically allocates or reclaims the provisioned memory of each tenant according to the miss ratio curve. With a two-phase memory resizing scheme, Genie resizes tenants' memory capacities to meet the hit ratio requirements.

Through the holistic design we present, Genie addresses the aforementioned challenges and achieves fine-grained, immediate, and on-demand resource scaling.

2) System Framework: Fig. 3 depicts the framework of Genie, which demonstrates the core components for fine-grained and immediate serverless caching. During the request processing, Genie utilizes a lightweight *Workload Profiler* to profile dynamic characteristics for workload-inspired resource resizing,

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

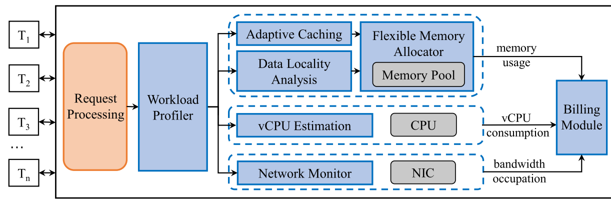

Fig. 3. System framework(T_1-T_n represent the tenants).

including skewness, average item size, GET/SET ratio, and reuse distance histogram. It decouples the memory into a shared *Memory Pool*, which is managed by *Flexible Memory Allocator*. The allocator resizes the memory capacity according to the miss ratio curve. Besides, it adjusts the eviction policy to improve memory efficiency through *Adaptive Caching*. In terms of computational resources, a machine learning model is implemented to estimate the CPU consumption of each tenant. As for network resources, it incorporates a *Network Monitor* to track the network traffic and occupancy. Moreover, we implement namespace isolation and access control mechanisms for the isolation of multi-tenancy. Based on the holistic design, Genie enables fine-grained, immediate, and flexible resource resizing and billing.

IV. Compute Elasticity

Elasticity in cloud computing requires the ability to adapt to time-varying workload pressures. For *Compute Elasticity*, Genie finds out the relationship between workload characteristics, throughput, and the usage of computational resources. It proposes a lightweight learned approach to measure CPU usage in fine granularity.

A. Workload Characterization

To determine which factors affect CPU usage in IMKVs, we study various workload characteristics and identify the following four characteristics. These characteristics are inherent properties of the request stream and have varying degrees of impact on system performance, providing insights into estimating the vCPU usage.

The skewness of the access distribution: Web accesses have been shown to follow an approximate Zipfian distribution [10], [12] with various skew factor α,

$$y_x = Cx^{-\alpha}, \quad (1)$$

where y_x is the access count of x-th most popular item, and α is the skew factor. A larger α indicates that most requests access fewer popular keys, and $\alpha = 0$ and 0.99 corresponds to uniform and skewed distribution, respectively. A highly skewed workload indicates good locality and fewer CPU cache misses, while a low skewness incurs more random memory accesses and significantly degrades system performance. As shown in Fig. 4(a), a highly skewed workload leads to a better throughput when α is close to 0.99 with 0.5 GET.

GET/SET ratio: In IMKVs, a GET operation retrieves the value for a given key if it exists; otherwise reloads the value into memory, and a SET operation adds a new item into the store or updates the value if it already exists. GET/SET ratio is the percentage of GET requests that occupies the overall operations. For instance, a 0.95 GET (0.05 SET) implies GET operations occupy 95% of tenants' requests, and SET operations occupy 5%. A larger GET/SET ratio implies fewer memory copies of the key, which leads to a higher throughput. As shown in Fig. 4(b), uniform and read-intensive workloads lead to a modest performance improvement (The server is network bottlenecked under large items with 0.95 GET).

Key size and value size: In IMKVs, items' key sizes and value sizes significantly affect the performance. Larger keys lead to apparent performance degradation, occupying more computational resources to process memory comparison when parsing the request and memory copy when storing items. As shown in Fig. 4(c), larger keys result in performance loss (value size is 8 bytes). For value sizes, a GET or SET operation will incur a memcpy to retrieve or store the value for sooner access, for which larger values occupy more computational resources and result in a performance decrease. The effect of value size is shown in Fig. 4(d) (key size is 8 bytes).

The characteristics analyzed above could characterize the impact of users' workloads. Under the same CPU utilization of the server, these characteristics have different impacts on system performance. Therefore, workloads with different characteristics will occupy varying numbers of computational resources under a stable request rate. It brings insight into achieving computational resource estimation through a machine-learning model.

B. The Model for Lightweight vCPU Usage Estimation

As the CPU consumption of each tenant depends on the characteristics of their workloads and request rates, Genie adopts regression models to estimate the CPU usage with fine granularity. Specifically, the request rate, which refers to the number of requests generated by the tenant and processed by the system, is equivalent to the tenant's throughput in the context of Genie's model. After the model has been trained and adopted in Genie, it predicts the vCPU usage by taking as input the workload characteristics (key size, value size, skewness, GET/SET ratio) and the tenant's throughput.

To ensure that users can purchase computational resources in a unified unit on different servers, Genie abstracts CPUs and provides tenants with a conception of virtual CPU quota, vCPU to represent a portion of the underlying physical CPU. The model's input includes workload characteristics we analyzed in Section IV-A and the maximum throughput of Genie under the corresponding workloads. The model's output is the number of vCPUs occupied by each tenant.

Through our custom key-value request generator that based on the YCSB benchmark [29], we create diverse workloads by varying the key size (8–128 bytes), value size (8–1024 bytes), GET/SET ratio (50% –95% GET), and the skew factor α (0–0.99). YCSB's popularity generation function is used to simulate different access skewness levels, while key and value sizes are selected to reflect typical web application scenarios.

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

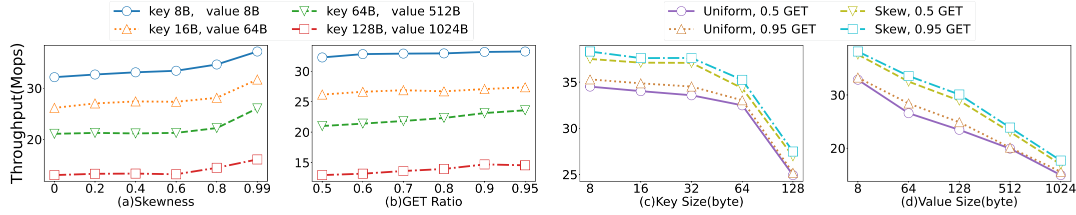

Fig. 4. End-to-end throughput under various workloads.

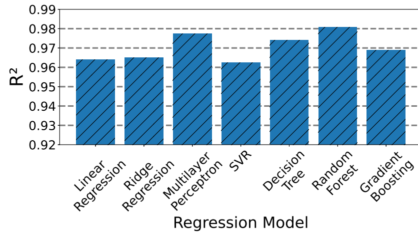

Fig. 5. R^2 of different models.

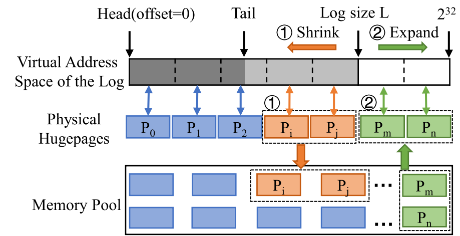

Fig. 6. Design of the flexible log-structured memory allocator.

After generating workloads, we adjust the number of vCPUs and measure the maximum meaningful throughput (with a 90% GET hit ratio) under each configuration.

The dataset, comprising 16,000 data points, was divided into training and test sets. We evaluated several regression models for their predictive accuracy, shown in Fig. 5. Among these models, we adopt the multilayer perceptron to predict vCPU consumption in Genie due to its high accuracy, low prediction cost, and ability to extrapolate values beyond the training set. With this model, Genie could achieve precise vCPU allocation and optimize resource utilization, which ultimately benefits both tenants and cloud vendors.

Through the machine learning model, we input the tenant's characteristics and throughput; then, we get the vCPU usage in real-time. The estimation cost we implemented in Genie is negligible. It takes less than 1 μs to make an estimation. After the estimation, the tenant's computational resource consumption could be billed under various billing policies with trivial overhead. When new CPU types are introduced, our model needs to be retrained to accommodate performance differences across CPUs for the same workload.

V. Memory Elasticity

To achieve fine-grained and prompt memory elasticity, Genie adopts an append-only log as the underlying data structure to store items. On top of the log-structured allocator, it proposes a flexible mechanism to resize the memory capacity through the analysis of data locality. Meanwhile, it proposes adaptive caching to improve memory efficiency.

A. Flexible Log-Structured Memory Allocator

Genie designs a flexible log-structured memory allocator to achieve memory elasticity, shown in Fig. 6. To optimize memory utilization, Genie proposes a pooled memory policy across tenants, and the memory pool is composed of 2 MiB hugepages (from P_0 to P_n in this figure), which reduces TLB misses and improves performance. To start with, each thread of Genie applies for a virtual address space for each tenant (e.g., 2^{32}). Then Genie allocates hugepages from the memory pool and maps them to the corresponding virtual addresses.

In Genie, each data partition of a tenant is a flexible log with in-place updates. *Head* points to the start (offset $= 0$), and *Tail* points to the location where new data items will be written. With the move of *Tail* brought by the newly stored items, old items will be overwritten sequentially in the order in which they were stored. When the log needs to expand, Genie calculates the number of required hugepages and requests available hugepages from the memory pool (P_m and P_n). After that, it maps them to the end of the log. When the log needs to shrink, Genie unmaps corresponding hugepages and reclaims them to the memory pool (P_i and P_j).

Hit ratio: is an important indicator. It has been reported that improving the cache hit ratio will greatly improve system performance [8], [12]. Genie aims to resize the memory according to users' hit ratio requirements. Since a miss (or hit) ratio curve depicts the relationship between the cache size and the miss ratio [41], Genie periodically constructs the miss ratio curve to guide the memory allocation of each tenant.

Miss Ratio Curve Construction: A miss ratio curve could be calculated from *reuse distance distribution* of the workload. *Reuse distance* for an item is measured as the number of other unique items since the last access to the item, e.g., the number of access to other unique items from the previous access to item A to the current access to item A. After capturing the *reuse distance distribution*, we calculate the miss ratio curve through the following equation, where *c* is the cache size, and the probability of reuse distance is acquired from the distribution:

$$\text{miss-ratio}(c) = 1 - \sum_{x=0}^{c} P(\text{reuse-distance} = x) \tag{2}$$

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

In Genie, we apply SHARDS [41] to obtain the reuse distance distribution, then calculate the miss ratio curve of an LRU cache through (2). SHARDS is a recent advancement in reuse distance analysis, which makes use of spatial sampling to track references and constructs miss ratio curves with the space complexity of O(1) and linear run time.

Flexible Memory Allocation: Upon a tenant's first arrival, Genie will allocate a predefined cache size, currently set to 20 MiB. Then it periodically resizes the memory of each tenant according to the miss ratio curve. The flexible memory allocation mechanism consists of two phases, *Direct-Fit* and *Fine-Grained Resizing*. Before memory resizing, Genie samples and collects item access-related information with a 0.1% sampling rate of *set sampling* [42].

1) Direct-Fit: After analyzing the current workload, Genie calculates the optimal log size L' for a desired hit ratio (e.g., 90%) using the MRC. If the new size is larger, Genie expands the log by allocating more hugepages from the memory pool(①). For shrinkage ($L' > L$), to prevent hit ratio fluctuations, Genie delays shrinking memory until the *Tail* points to the next hugepage. This ensures that shrinking removes rarely revisited items rather than popular ones (②). Additionally, Genie prioritizes reclaiming memory from the log's *Head*, where expired items tend to accumulate due to the replacement strategy that reinserts popular items at the *Tail*. By targeting underutilized regions, this approach reduces fragmentation and improves memory efficiency. The shrinking process lasts at most two rounds to ensure stable performance, with the reclaimed hugepages returned to the memory pool. This approach ensures that memory reclamation targets regions with higher levels of underutilization, effectively reducing fragmentation and improving memory efficiency.

2) Fine-Grained Resizing: Direct-Fit may not always precisely meet a tenant's target hit ratio due to SHARDS' median MAE of 2% at a 0.1% sampling rate and potential workload shifts. Additionally, the log's sequential writes and replacement strategy impact memory efficiency. To overcome this, Genie implements *Fine-Grained Resizing*, adjusting log size to bridge the gap between actual and target hit ratios, and refining the memory allocation until the target is achieved. Specifically, we incrementally adjust the corresponding log in 2MiB units (consistent with the size of a hugepage). The added hugepage is placed immediately after the one currently pointed to by *Tail*, while the memory to be reduced is the one following the hugepage pointed to by *Tail*. Besides, when a tenant does not access the Genie for an extended period and all its items have expired, Genie will reclaim the memory resources allocated to that tenant and maintain records of their historical cache sizes. Upon resuming activity, the newly allocated cache size is set to match their previous usage.

Overall, Genie employs a flexible memory allocator with append-only logs to store items and a shared pool for resource sharing. It designs a dynamic mechanism to periodically adjust memory capacities based on miss ratio curves, which meet the demand of hit ratios while maintaining stable performance.

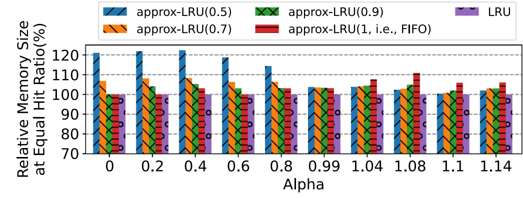

Fig. 7. Relative memory sizes under various thresholds of approx-LRU.

B. Adaptive Caching

Genie studies the inefficiencies of a log-structured memory allocator and designs an adaptive caching to improve memory efficiency under different workloads.

To keep popular items in memory, MICA, a high-performance IMKV that adopts append-only logs for item storage, proposes an approximate LRU (approx-LRU for short) that reinserts frequently visited items. It reinserts the currently accessed item while its position is more than half the entire log from *Tail*. However, the approx-LRU it proposed will result in varying inefficiencies under different workloads, as popular items may occupy two positions in the log. Specifically, under a skewed distribution, only a small proportion of items are frequently visited and will be reinserted into the log, which leads to a slight inefficiency. Under a uniform distribution, items are almost equally accessed, and a large portion occupies two positions, resulting in apparent memory waste.

Recent studies have shown FIFO and LRU perform similarly in hit ratios according to real-world traces [10], [43]. Motivated by their comparable effectiveness and the inefficiencies of approx-LRU, we examined memory usage with a 90% hit ratio across various workloads and tiny items, focusing on different approx-LRU thresholds. These thresholds determine when to reinsert items based on their distance from *Tail*. For instance, with a 0.7 threshold, items are reinserted if they're beyond 70% of the log's length from *Tail*. Notably, a 1 threshold equates to FIFO, whereas MICA uses 0.5 as the threshold. We excluded thresholds below 0.5 to avoid the inefficiency of popular items occupying multiple positions.

Fig. 7 depicts relative memory sizes under different thresholds of approximate LRU. The memory size calculated from SHARDS is the baseline, representing the ideal log size with 100% efficiency. As shown in Fig. 7, different thresholds of approx-LRU require different amounts of memory to achieve the same hit ratio. When the skew factor $\alpha < 1$, FIFO is closest to the ideal memory size, and when $\alpha > 1$, approx-LRU with threshold 0.5 is most comparable to LRU.

To improve memory efficiency under various and time-varying workloads, we propose an adaptive caching policy. It dynamically adjusts the eviction policy according to the workload's skewness, a low skewness ($\alpha < 1$) using FIFO and a high skewness ($\alpha > 1$) using approx-LRU with threshold 0.5. The adaptive eviction policy is memory efficient regardless of various skewness of workloads. It considers factors including access frequency, item size, and memory efficiency.

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

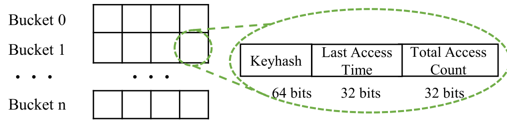

Fig. 8. Design of the lossy hash table for data locality analysis.

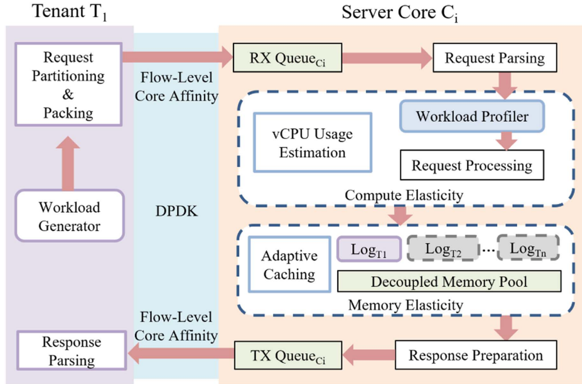

Fig. 9. Genie's architecture.

Overall, Genie proposes an adaptive mechanism for evicting items according to workload distributions dynamically. It improves the memory efficiency of each tenant and contributes to resizing memory efficiently and effectively.

C. Genie Implementation

1) Workload Profiler: Genie designs a lightweight workload profiler to profile dynamic workload characteristics. As shown in Fig. 8, a lossy hash table is designed for data locality analysis. Each hash entry consists of a *keyhash*, a *last access time*, and a *total access count* of the item. Once an item is sampled, Genie will update the hash table to track the workload. The *last access time* of sampled items is used to update the reuse distance histogram for *Memory Elasticity*, and the *total access count* is used for estimating the skewness.

The skew factor α is derived using Least-Squares Regression (LSR) [44] on popular items, as it forms a straight line in a logarithmic scale with slope α. A R^2 value over 0.8 is considered a good fit for the Zipf distribution. Additionally, a profiler tracks each tenant's average key size, value size, and `GET/SET` ratio for *Compute Elasticity*.

2) System Implementation and Architecture: In our experiments, we first evaluated Genie on a single machine and then extended the setup to include two server nodes and two client nodes, analyzing its scalability across these configurations.

In Genie, each tenant is authenticated with a unique namespace, and each of their requests is routed to a specific CPU using a hash function. Each CPU handles a partition of the tenant's requests and stores the corresponding data in its memory. Memory partitions are independently resizable based on workload demands, and overall CPU usage is tracked per tenant. This design ensures high performance through data partitioning while maintaining strong tenant isolation via namespace authentication and independent memory spaces.

Fig. 9 illustrates the system architecture, highlighting key components along with control and data flows. Genie ensures resource isolation and efficient data handling through flow-level core affinity, decoupled memory management, and workload-aware elasticity. Flow-level core affinity directs tenant traffic to dedicated server cores, reducing context switching and enhancing CPU cache efficiency. Tenants' items are stored in dynamically resizable logs from a decoupled memory pool, enabling efficient workload-driven memory elasticity. In Genie, each CPU core is bound to dedicated RX/TX queues, enabling fast packet processing with low latency. This design achieves high throughput, low latency, and scalable resource management. The source code can be accessed at https://github.com/Yangsx-1/Genie.

VI. Evaluation

In this section, we evaluate fine-grained and prompt elasticity, multi-tenancy, throughput, and latency in Genie under various workloads and configurations.

A. Evaluation Setup

Genie, developed in C++, starts with MICA [39] but significantly evolves with a redesigned architecture to emphasize its serverless features. For all workloads, a 1 GiB hash table is used to index key-value items of each worker thread. For each partition of the tenant, Genie uses a 16 MiB hash table for data locality. Besides, it uses the open-source DPDK [45] to support high-speed I/O.

1) Server/Client Configuration: To evaluate Genie's effectiveness and lightweight design, we developed a prototype deployed on individual bare metal machines, simplifying our experiments to focus on serverless caching. The Genie server runs on a machine with an Intel Xeon E5-2695 v4 Broadwell processor with 18 cores, 2 NUMA nodes, and 45 MiB L3 cache, running at 2.1 GHz. The server is equipped with 128 GiB DDR4 DRAM and two 100 Gbps Ethernet ports. The server uses eight cores, and reserves 42 GiB for hugepages, with each core processing requests through a dedicated worker thread and one additional thread for cleanup duties. The client employs an Intel Xeon Gold 6140 processor with 18 cores, a single NUMA node, and 24.8 MiB L3 cache at 2.3 GHz, complemented by 128 GiB DDR4 DRAM and a 100 Gbps Ethernet port.

2) Baseline: We use two baselines in experiments for comparison. First, we use MICA-r, a custom version of MICA [39], as one of our baseline for its high performance of key-value caching. It allows a single thread to clean up *stale pointers* of hash tables that were previously held by each worker thread, for the binding between the width of the log and the scanning speed limits the scalability of the memory, which is consistent with Genie. Besides, we add a `RELOAD` operation after a `GET MISS` to restore the item into the system, consistent with research on memory estimation for caches [41]. Therefore, MICA-r has the same underlying IMKV structure as Genie but does not have serverless modules.

Second, we use InfiniCache [16], a state-of-the-art serverless in-memory key-value caching atop serverless functions, as the other baseline. To ensure a fair comparison, we deployed

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

TABLE I
THREE DATASETS FOR EVALUATION

Dataset	Key Size(B)	Value Size(B)	Count
Tiny	8	8	256Mi
Small	16	64	128Mi
Large	128	1024	8Mi

InfiniCache on the same physical machine used for Genie's experiments and allocated them with the same resources. This approach ensures fairness by addressing the significant differences in resource allocation and management between Genie, which can be deployed on bare-metal machines or virtual machines, and InfiniCache, which relies on deployment across multiple serverless functions.

To capture the optimal performance of InfiniCache, we adhere to the original design and setup outlined in the InfiniCache paper while exploring additional configurations for comparison. Specifically, InfiniCache employs erasure coding $(d+p)$ to encode a key-value item into multiple shards (d data shards and p parity shards). These shards are distributed across different serverless functions. Following the original configuration, we evaluate three variants of InfiniCache to explore the best performance: InfiniCache-(8+0), (7+1), and (6+2). In each variant, a key-value item is divided into 8 shards, with varying levels of redundancy provided by the parity shards. These shards are cached separately, requiring reconstruction during retrieval. Besides, we evaluate a modified configuration, InfiniCache$*$, in which each item is stored as a whole without being divided into shards. This eliminates the need for the client to wait for all shards to arrive and reconstruct the item. It enables parallel processing of requests and leads to improved performance compared to the original design.

3) Workloads and Datasets: We evaluate Genie with YCSB synthetic workloads [29] and real-world key-value traces from Twitter [10], [46]. Both of them have been widely used in evaluations of key-value stores [1], [39], [47], [48]. For YCSB synthetic workloads, Genie clients adopt three kinds of datasets (shown in Table I) and approximated Zipf distribution algorithms [49], [50] to generate the working set.

To evaluate the elasticity of CPU and memory resources, we use synthetic time-varying workloads with dynamic characteristics that have been proven to change over time in Section II-A. For studies on performance isolation of multi-tenancy, system throughput, and latency, we use two workload types: *Uniform* and *Skew*, which is in line with YCSB [29] and has been widely adopted in evaluations of KVs [39], [48]. Besides, to evaluate the effectiveness of the adaptive eviction policy, we use two Twitter traces [10], [46], namely clusters 24 and 48 that have been used in recent cache studies [1].

B. Fine-Grained vCPU Estimation

In this subsection, we aim to assess the effectiveness of vCPU estimation in handling diverse and dynamic workloads. We use a synthetic workload with dynamic characteristics and request rates and compare the system throughput and vCPU usage between MICA-r and Genie.

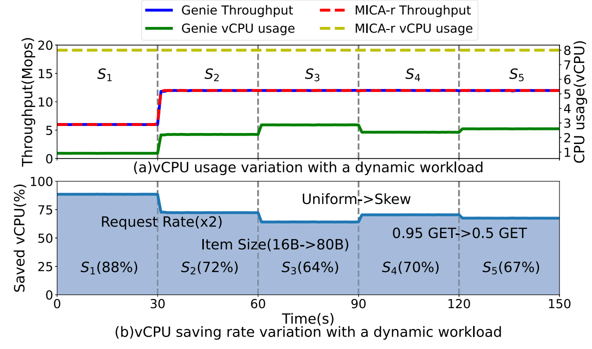

Fig. 10. vCPU estimation under a dynamic workload.

As shown in Fig. 10, the evaluation of workload dynamism encompasses five distinct stages, denoted as S_1 to S_5, with each stage's workload being modified based on the preceding stage. S_1 represents a 128 MiB uniform *Tiny* workload with 0.95 GET, and the request rate is 6 Mops.

As depicted in Fig. 10(a), the *Compute Elasticity* in Genie is reactive and incurs little performance degradation compared to MICA-r. Genie exhibits significant fluctuations in vCPU usage across different stages rather than the constant vCPU usage (8 vCPUs in this experiment) in MICA-r. The transition of the doubled request rate from S_1 to S_2 leads to the vCPU usage increasing from 0.92 to 2.2, which follows the CPU scalability of Genie. From S_2 to S_3, the item size grows from *Tiny* to *Small*, and the vCPU usage grows to 2.87 as large items occupy more computational resources during the request processing. From S_3 to S_4, the workload distribution changes from uniform to 0.99 skewed, leading to a vCPU usage change from 2.87 to 2.36. Lastly, the transition from S_4 to S_5 changes the workload from 0.95 GET to 0.5 GET, contributing to the vCPU usage changing from 2.36 to 2.6. Meanwhile, its ability to promptly adapt to workload changes across different stages and accurately estimate CPU resource usage in real time underscores the exceptional efficiency of Genie's workload profiler and prediction model.

Fig. 10(b) exhibits the vCPU saving rates under the dynamic workload compared to MICA-r. From S_1 to S_5, the saving rates range from 64% to 88%, which is very considerable compared to current cloud IMKVs.

C. Workload-Inspired Memory Elasticity

In this subsection, we demonstrate the efficiency and effectiveness of memory elasticity. It resizes the memory capacity of each tenant under the hit ratio requirement and optimizes memory efficiency under various workloads in real-time.

First, we evaluate Genie's memory elasticity using a synthetic workload with changing skewness and item sizes. Fig. 11 shows the memory usage and hit ratio variation under the dynamic workload. The dynamism comprises three stages, each representing a distinct working set of ten minutes. $S_1{}'$ is a highly skewed workload ($\alpha = 1.1$) on small items, $S_2{}'$ is a uniform

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

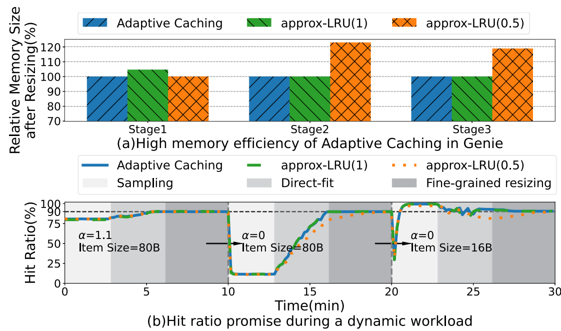

Fig. 11. Memory resizing under a dynamic workload.

workload on small items, and $S_3{}'$ is a uniform workload on tiny items. The initial size of each log is 20 MiB. At the beginning of $S_1{}'$, the hit ratio is about 80% as a highly skewed workload requires less space for popular items. After sampling, Genie captures the locality of the workload. It calculates the ideal memory size and the skew factor α. Then, it resizes the memory to meet the hit ratio requirement and adjusts the eviction policy to improve efficiency.

As shown in Fig. 11(a), after flexible memory allocation, Genie achieves memory saving under different working sets and item sizes (using adaptive caching as the baseline). In $S_1{}'$, FIFO requires about 5% extra memory capacity to meet the hit ratio requirement compared to adaptive caching, and in $S_2{}'$ and $S_3{}'$, the approx-LRU(0.5) requires around 23% and 19% extra space compared to adaptive caching, respectively.

Fig. 11(b) shows the changes in hit ratios under the dynamic workload. The *Fine-Grained Resizing* of each stage enables the hit ratio to reach the target (i.e., 90%) within a few minutes. From $S_1{}'$ to $S_2{}'$, as the skew factor α changes from 1.1 to 0, there is a drastic drop in hit ratio because more memory is required to cache popular items. From $S_2{}'$ to $S_3{}'$, the item size shifts from *Small* to *Tiny*, for which the hit ratio descends a lot and reaches almost 100% quickly. The shrink of the log from *Direct-Fit* of $S_2{}'$ leads to a hit ratio trembling of no more than 5%. Finally, the hit ratio remains stable at 90%.

Second, we evaluate the effectiveness of memory elasticity in Genie under Twitter workloads, including cluster 48 with α of 0.8191 and cluster 24 with α of 1.3726, respectively. As shown in Fig. 13, Genie improves memory efficiency regardless of the various skewness of workloads.

D. System Throughput and Latency

In this subsection, we evaluate and compare the end-to-end throughput (maximum throughput with 90% `GET` hit ratio) and the latency distribution of Genie with MICA-r and InfiniCache. The interval of vCPU estimation is one second.

Comparison with MICA-r: We compare the throughput and latency distributions of Genie with MICA-r under various workloads from Table I with different `GET`/`SET` ratios and skewness. As is shown in Fig. 12(a)-(c), We find that the throughput is rarely affected by *compute* and *memory elasticity* of Genie. It performs only 1% -2% degradation with *compute* and *memory elasticity* under various workloads. Besides, as shown in Fig. 12(d), with only *compute elasticity*, the distribution exhibits almost the same as that of MICA-r, except for the tail latency varies from 84 to 92 μs. For Genie with *compute* and *memory elasticity*, the average latency is about 1-2 μs higher than MICA-r, which is imperceptible for users, and the tail latency grows to 98 μs.

The main source of performance overhead in Genie is the workload profiler, which samples and computes workload characteristics in real time. With a sampling rate under 0.1%, it introduces roughly 1% -2% overhead.

Comparison with InfiniCache: We further compare and evaluate the end-to-end throughput and latency distributions of Genie and the baseline InfiniCache on uniform 50% `GET` workloads with workloads from Table I.

Specifically, Fig. 15(a) depicts the throughput of Genie and InfiniCache under different workloads and configurations. As shown in the figure, under the original erasure coding configuration, the throughput of InfiniCache for tiny, small, and large items is 28, 22, and 17 operations per second, respectively, which represent the best performance achieved across three different configurations (InfiniCache-(8+0), (7+1), and (6+2)). Without erasure coding, the throughput of InfiniCache (InfiniCache$*$ in the figure) increases significantly to 122, 107, and 90 operations per second, respectively. However, even with this improvement, its performance falls far short of Genie. As shown in this figure, Genie achieves around 35, 28, and 13 million operations per second (Mops) for tiny, small, and large items, respectively.

Fig. 15(b)-(d) illustrates the end-to-end latency distributions of Genie and InfiniCache. The results show that InfiniCache incurs significant end-to-end request latency across different scenarios, with most values ranging from 20 to 100 ms under its original erasure coding configurations (InfiniCache-(8+0), (7+1), and (6+2)). Even without erasure coding, the latency of InfiniCache (InfiniCache$*$ in this figure) remains between 10 and 60 ms, which is still much higher than Genie's. However, as shown in the figure, Genie consistently maintains an end-to-end latency of less than 1 ms even under large items.

We further examine the performance of Genie and InfiniCache with item sizes from 4 KiB to 4 MiB, which are significantly larger than those typically encountered in the web applications we focused on. We use a uniform 100% `GET` workload, with results shown in Fig. 16. In this experiment, Genie continues to outperform InfiniCache in both throughput and latency distributions. Specifically, Genie achieves 6.4 Mops for 4 KiB items, and 3218 ops for 4 MiB items, which are around 5 and 2.8 orders of magnitude higher than InfiniCache's throughput (29 ops for 4 KiB items and 5 ops for 4 MiB items). As for latency, Genie delivers an average latency of around 40 μs, 250 μs, and 2.5 ms with 4 KiB, 512 KiB, and 4 MiB item sizes. For InfiniCache, the average latency is around 38 ms, 70 ms, and 184 ms, which is lower than InfiniCache about 98.64% to 99.89%.

The significant performance improvement of Genie over InfiniCache, especially for small items, can be attributed to two main factors. First, InfiniCache relies on serverless functions for dynamic resource allocation, which are built on virtualized instances. These virtualized instances introduce inherent overhead and significant invocation latency, which reduce system throughput and limit InfiniCache's efficiency in handling small

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

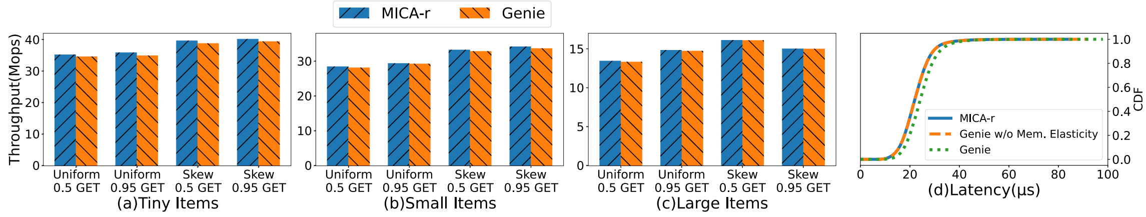

Fig. 12. Comparison on end-to-end throughput and latency distribution between MICA-r and Genie.

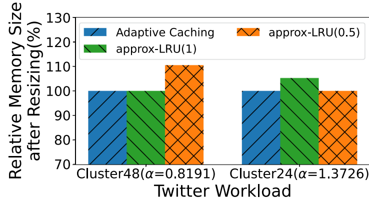

Fig. 13. Memory efficiencies of adaptive caching.

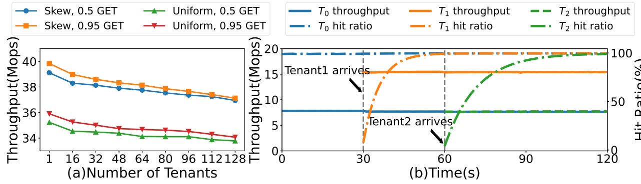

Fig. 14. System throughput and performance isolation with multiple tenants.

items. Second, serverless functions could be reclaimed by cloud providers at any time, resulting in potential item loss. To address this, InfiniCache implements a fault-tolerance mechanism that shards items into multiple data and parity shards, which are distributed across different serverless functions. To retrieve an item, InfiniCache must fetch these shards, decode them, and reconstruct the original item, introducing extra overhead. In contrast, Genie introduces a dynamic resource allocation mechanism with real-time workload profiling and resource modeling. It avoids the virtualization overhead and the need for item sharding and reconstruction, enhancing overall system efficiency. With this approach, Genie achieves microsecond-level response times with only a 1% –2% performance overhead for small items.

E. Resource Savings

In this subsection, we evaluate the resource savings of Genie compared to cloud IMKVs under W_1 to W_6 workloads. Fig. 17 shows the relative resource savings of Genie, using instances `P` and `Q` of `ecs.u1` from AsparaDB for Memcache as the baseline. For workloads W_1 to W_6, Genie saves an average of 58.3% CPU and 49.9% memory.

F. Multi-Tenancy

In this subsection, we evaluate the ability of multi-tenancy and its impact on performance. It achieves lightweight serverless caching with per-tenant performance isolation.

First, we measure the system's overall throughput with increasing tenants and demonstrate that the multi-tenancy introduces negligible overhead. Fig. 14(a) depicts the throughput as a function of the number of tenants under uniform and skewed workloads of 0.5 `GET` and 0.95 `GET` on tiny items. The performance degradation is only about 1.5 to 3 Mops (4% -6%) under four workloads with 128 tenants, which evidences Genie is a lightweight infrastructure for serverless caching.

Second, we evaluate Genie's performance isolation as multiple tenants access the server concurrently. As shown in Fig. 14(b), when T_1 and T_2 join at 30 s and 60 s, respectively, T_0's throughput and hit ratio remain unaffected. This demonstrates Genie's ability to efficiently handle bursty tenant arrivals, with its workload profiler and prediction model quickly adapting to new workloads while maintaining stable performance for existing tenants.

G. vCPU Estimation Interval

This subsection shows Genie achieves vCPU estimation at millisecond granularity with minimal overhead. Fig. 20 shows the overall throughput and the latency distribution with various intervals of vCPU measurement on *Tiny* items. The estimation has a trivial impact on performance when the estimation interval $\geq 10^{-4}$ s (0.1ms). The end-to-end throughput degrades around 7% and 10% under uniform and skewed workloads when interval $= 10^{-6}$s (1 μs). Moreover, an interval $\geq 10^{-3}$s has minimal impact on the latency distribution. When interval $= 10^{-4}$s, it brings about a poor tail latency, and when the interval is smaller than 10^{-4}s, the median and upper quartile latency values significantly increase.

H. Scalability

Fig. 21(a)-(c) depicts Genie's throughput with two server nodes (Genie$*$ refers to Genie with two server nodes). The throughput of Genie$*$ is consistently around 1.9 times higher than that of Genie across different workloads. Besides, Fig. 21(d) depicts the end-to-end throughput with varying numbers of tenants from two client nodes. As shown in this figure, Genie's multi-tenant performance under different workloads, with a single server node and two client nodes, is consistent with the multi-tenant performance presented in Fig. 14. Due to resource constraints, our experiments were limited to three physical servers, but the observed linear scalability indicates that Genie can scale to larger deployments. Addressing challenges like distributed coordination and data consistency will be critical for future extensions.

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

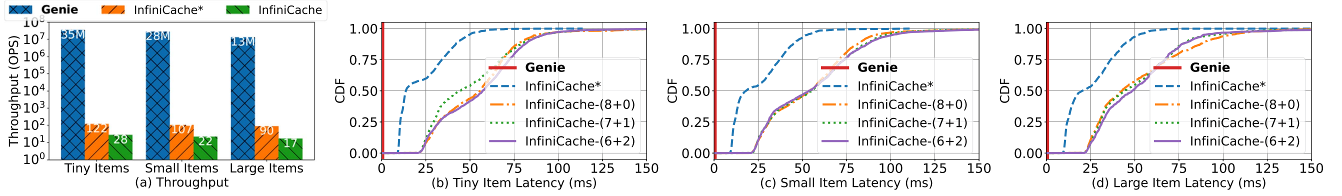

Fig. 15. Comparisons of throughput and end-to-end latency distributions between Genie and InfiniCache under different workloads and configurations.

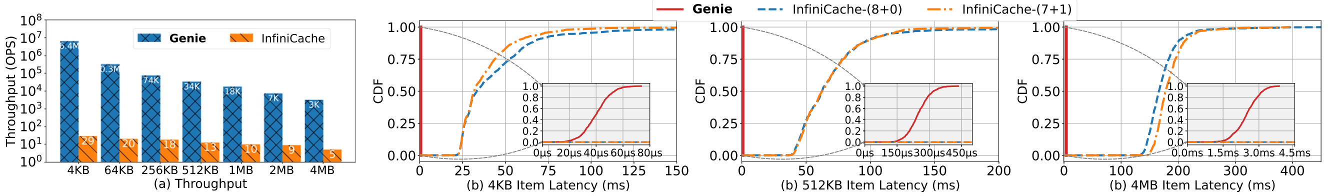

Fig. 16. Comparisons of throughput and end-to-end latency distributions between Genie and InfiniCache with value sizes from 4 KiB to 4 MiB.

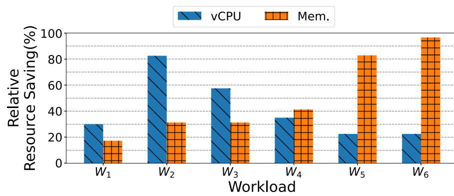

Fig. 17. Resource savings of Genie.

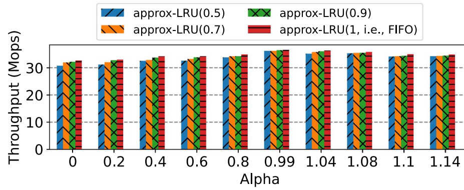

Fig. 18. Impact of adaptive caching thresholds.

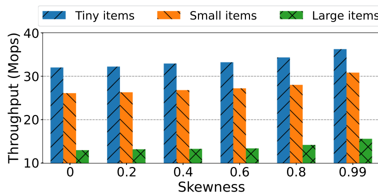

Fig. 19. Skewness impact on throughput.

I. Impact of Adaptive Caching Thresholds on Performance

Fig. 18 depicts the system throughput under various thresholds of approx-LRU. It can be observed that under different workloads, the larger the approx-LRU threshold, the higher the system's throughput. The throughput with a threshold of 1 is typically 1–2 Mops higher than that with a threshold of 0.5. This is because the smaller the thresholds, the more frequently popular items will be reinserted to the *Tail* of the log, which degrades the overall performance.

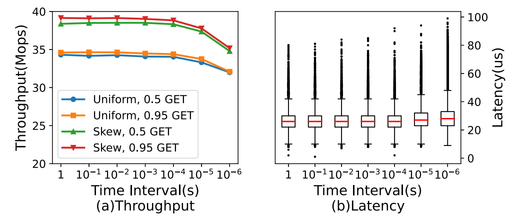

Fig. 20. End-to-end throughput and latency distributions under different intervals of vCPU estimation.

J. Impact of Workload Skewness on Performance

As shown in Fig. 19, we evaluated Genie's end-to-end throughput across varying levels of workload skewness (α). The results clearly show that the throughput increases with higher α values across all three kinds of workloads. This increase is due to a highly skewed workload, which improves data locality and reduces CPU cache misses, thereby enhancing Genie's overall performance.

VII. Related Work

Memory-Disaggregated key-value stores: Current memory-disaggregated KVs focus on either enhancing performance [26] or executing various caching algorithms [25]. However, the elasticity and efficiency they achieve are constrained due to the thread-level computational allocation and the inefficient slab-based memory allocator. Considering the long-tailed distribution of resource demands on the cloud [38], the coarse-grained and inefficient allocation leads to a significant number of users paying for resources that are allocated but unused. As a lightweight infrastructure, Genie could achieve better resource elasticity on disaggregated memory.

Resource efficiency and isolation: Prior work has improved resource efficiency and isolation in IMKVs through various policies. Segcache [1] optimizes TTL management to enhance memory efficiency, while LHD [2] prioritizes evictions based on expected storage contribution. Unlike these, Genie emphasizes resource elasticity, achieving flexible memory allocation and

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

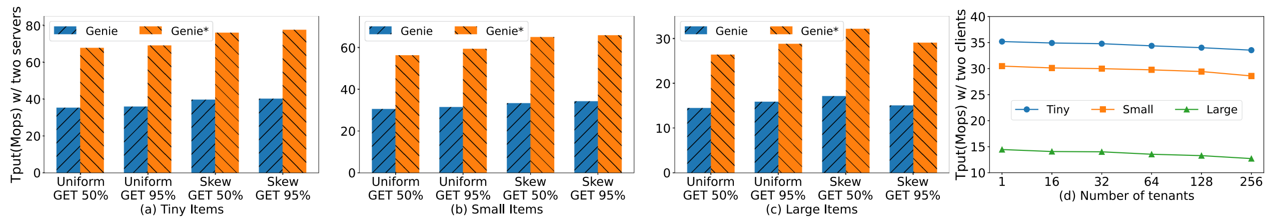

Fig. 21. Comparison of end-to-end throughput with different numbers of server and client nodes.

adaptive caching. For compute resources, SQLVM [51] measures task runtime on available cores to manage CPU utilization in relational databases, but this method incurs high overhead for high-performance IMKVs (shown in Fig. 2).

Dynamic memory management: Several studies focus on dynamically managing the memory of applications to optimize the hit ratio [8]. They adopt cache locality modeling algorithms [41] to estimate the miss ratio curve and adjust the memory accordingly. However, the slab-structured memory allocator they adopted limits system performance and multi-tenancy [1], [8]. Instead, Genie adopts append-only logs and proposes corresponding algorithms to improve memory efficiency and flexibility.

Several works focus on resource allocation and memory management in serverless computing. Cuki [52] estimates working set sizes and cache miss ratio curves but is incompatible with log-structured memory allocators. OFC [53] uses machine learning to predict memory demands but struggles with dynamic workloads due to its reliance on per-application models and coarse-grained adjustments. Faa$T [54] integrates CPU scaling with memory allocation but couples these resources, leading to coarse-grained elasticity and underutilization. Faasm [55] enables memory sharing for stateful functions but lacks fine-grained and dynamic CPU allocation.

Resource allocation of serverless functions: Several works have explored resource allocation and management for serverless functions. Freyr [36] uses deep reinforcement learning to reallocate over-provisioned serverless function resources, improving utilization. INFless introduces an ML domain-specific serverless platform with unified and heterogeneous resources. Rethinking [37] evaluates static CPU and memory configurations to optimize performance and cost but lacks the adaptability to runtime fluctuations. However, the inherent invocation latency of serverless functions reflects a fundamental limitation, making it challenging to support latency-sensitive in-memory key-value caching.

VIII. Conclusion

This paper introduces Genie, a lightweight serverless infrastructure for in-memory key-value caching, featuring fine-grained and prompt elasticity. Genie employs a novel approach that enables dynamic and independent resource allocation of multiple tenants. It adopts a lightweight machine learning approach to estimate vCPU consumption at ms level. Not only that, Genie dynamically adjusts the allocated memory based on estimations of each tenant's working set size at the megabyte level. Our experiments validate that Genie's implementation of a serverless, in-memory key-value caching system not only delivers lightweight and fine-grained resource elasticity but also significantly outpaces existing solutions.

References

[1] J. Yang, Y. Yue, and R. Vinayak, "Segcache: A memory-efficient and scalable in-memory key-value cache for small objects," in *Proc. 18th USENIX Symp. Netw. Syst. Des. Implementation*, 2021, pp. 503–518.
[2] N. Beckmann, H. Chen, and A. Cidon, "{LHD}: Improving cache hit rate by maximizing hit density," in *Proc. 15th USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 389–403.
[3] E. Jonas et al., "Cloud programming simplified: A berkeley view on serverless computing," 2019, *arXiv: 1902.03383*.
[4] A. Fox et al., "Above the clouds: A berkeley view of cloud computing," Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, 2009.
[5] AWS, "Dynamodb accelerator," 2023. [Online]. Available: https://aws.amazon.com/dynamodb/dax/
[6] AWS, "Dynamodb pricing," 2023. [Online]. Available: https://aws.amazon.com/dynamodb/pricing/on-demand/
[7] D. Carra, G. Neglia, and P. Michiardi, "Elastic provisioning of cloud caches: A cost-aware TTL approach," *IEEE/ACM Trans. Netw.*, vol. 28, no. 3, pp. 1283–1296, Jun. 2020.
[8] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman, "Memshare: A dynamic multi-tenant key-value cache," in *Proc. 2017 USENIX Annu. Tech. Conf.*, 2017, pp. 321–334.
[9] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic modeling of data eviction in cache," in *Proc. 2016 USENIX Annu. Tech. Conf.*, 2016, pp. 351–364.
[10] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 191–208.
[11] C. Wang, A. Gupta, and B. Urgaonkar, "Fine-grained resource scaling in a public cloud: A tenant's perspective," in *Proc. IEEE 9th Int. Conf. Cloud Comput.*, 2016, pp. 124–131.
[12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. Meas. Model. Comput. Syst.*, 2012, pp. 53–64.
[13] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 209–223.
[14] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–32, 2022.
[15] AWS, "Amazon elasticache serverless for redis and memcached is now available," 2023. [Online]. Available: https://aws.amazon.com/blogs/aws/amazon-elasticache-serverless-for-redis-and-memcached-now-generally-available/
[16] A. Wang et al., "InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 267–281.
[17] AWS, "Introducing amazon elasticache serverless," 2023. [Online]. Available: https://www.youtube.com/watch?v=YYStP97pbXo&ab_channel=AWSEvents
[18] M. Labib, "Amazon elasticache deep dive," 2020. [Online]. Available: https://pages.awscloud.com/rs/112-TZM-766/images/Session%201%20-%20ElastiCache-DeepDive_v2_rev.pdf

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

[19] A. Singla, B. Chandrasekaran, P. B. Godfrey, and B. Maggs, “The internet at the speed of light,” in *Proc. 13th ACM Workshop Hot Top. Netw.*, 2014, pp. 1–7.
[20] J. Zhang et al., “Infinistore: Elastic serverless cloud storage,” in *Proc. VLDB Endow.*, vol. 16, no. 7, pp. 1629–1642, Mar. 2023.
[21] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, “Peeking behind the curtains of serverless platforms,” in *Proc. 2018 {USENIX} Annu. Tech. Conf.*, 2018, pp. 133–146.
[22] AWS, “AWS lambda,” 2023. [Online]. Available: https://aws.amazon.com/lambda/
[23] AWS, “Elasticache,” 2023. [Online]. Available: https://aws.amazon.com/elasticache/
[24] Alibaba, “ApsaraDB,” 2023. [Online]. Available: https://www.alibabacloud.com/product/apsaradb-for-memcache
[25] J. Shen et al., “Ditto: An elastic and adaptive memory-disaggregated caching system,” in *Proc. 29th Symp. Operating Syst. Princ.*, 2023, pp. 675–691.
[26] J. Shen et al., “{FUSEE}: A fully{ Memory-Disaggregated}{Key-Value} store,” in *Proc. 21st USENIX Conf. File Storage Technol.*, 2023, pp. 81–98.
[27] S.-Y. Tsai, Y. Shan, and Y. Zhang, “Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated {Key-Value} stores,” in *Proc. 2020 USENIX Annu. Tech. Conf.*, 2020, pp. 33–48.
[28] S. Chatterjee, M. Jagadeesan, W. Qin, and S. Idreos, “Cosine: A cloud-cost optimized self-designing key-value storage engine,” in *Proc. VLDB Endowment*, vol. 15, no. 1, pp. 112–126, 2021.
[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[30] J. Guo et al., “Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces,” in *Proc. IEEE/ACM 27th Int. Symp. Qual. Service*, 2019, pp. 1–10.
[31] A. Mahgoub et al., “Sophia: Online reconfiguration of clustered NoSQL databases for time-varying workloads,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 223–240.
[32] J. Chen et al., “HotRing: A hotspot-aware in-memory key-value store,” in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 239–252.
[33] I. Baldini et al., “Serverless computing: Current trends and open problems,” in *Research Advances in Cloud Computing*. Berlin, Germany: Springer, 2017, pp. 1–20.
[34] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman, “Rocksteady: Fast migration for low-latency in-memory storage,” in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 390–405.
[35] X. Qin, W. Zhang, W. Wang, J. Wei, X. Zhao, and T. Huang, “Optimizing data migration for cloud-based key-value stores,” in *Proc. 21st ACM Int. Conf. Inf. Knowl. Manage.*, 2012, pp. 2204–2208.
[36] H. Yu, H. Wang, J. Li, X. Yuan, and S.-J. Park, “Accelerating serverless computing by harvesting idle resources,” in *Proc. ACM Web Conf.*, 2022, pp. 1741–1751.
[37] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, “With great freedom comes great opportunity: Rethinking resource allocation for serverless functions,” in *Proc. 18th Eur. Conf. Comput. Syst.*, 2023, pp. 381–397.
[38] C. Loboz, “Cloud resource usage—heavy tailed distributions invalidating traditional capacity planning models,” *J. Grid Comput.*, vol. 10, pp. 85–108, 2012.
[39] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, “MICA: A holistic approach to fast in-memory key-value storage,” in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 429–444.
[40] Memcached, “Memcached,” 2023. [Online]. Available: https://www.memcached.org/
[41] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, “Efficient {MRC } construction with { SHARDS},” in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 95–110.
[42] X. Hu et al., “Fast miss ratio curve modeling for storage cache,” *ACM Trans. Storage*, vol. 14, no. 2, pp. 1–34, 2018.
[43] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat, “It’s time to revisit LRU versus FIFO,” in *Proc. 12th USENIX Conf. Hot Top. Storage File Syst.*, 2020, pp. 12–12.
[44] Y. Yang and J. Zhu, “Write skew and zipf distribution: Evidence and implications,” *ACM Trans. Storage*, vol. 12, no. 4, pp. 1–19, 2016.
[45] Intel, “Data plane development kit,” 2023. [Online]. Available: https://www.dpdk.org/
[46] Twitter, “Twitter cache trace,” 2020. [Online]. Available: https://github.com/twitter/cache-trace
[47] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, “Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores,” in *Proc. VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015.
[48] Z. Qiu et al., “FrozenHot cache: Rethinking cache management for modern hardware,” in *Proc. 18th Eur. Conf. Comput. Syst.*, 2023, pp. 557–573.
[49] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, “Quickly generating billion-record synthetic databases,” in *Proc. 1994 ACM SIGMOD Int. Conf. Manage. Data*, 1994, pp. 243–252.
[50] W. Hörmann and G. Derflinger, “Rejection-inversion to generate variates from monotone discrete distributions,” *ACM Trans. Model. Comput. Simul.*, vol. 6, no. 3, pp. 169–184, 1996.
[51] S. Das, V. R. Narasayya, F. Li, and M. Syamala, “CPU sharing techniques for performance isolation in multi-tenant relational database-as-a-service,” in *Proc. VLDB Endowment*, vol. 7, no. 1, pp. 37–48, 2013.
[52] R. Gu et al., “Adaptive online cache capacity optimization via lightweight working set size estimation at scale,” in *Proc. 2023 USENIX Annu. Tech. Conf.*, 2023, pp. 467–484.
[53] D. Mvondo et al., “OFC: An opportunistic caching system for FaaS platforms,” in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 228–244.
[54] F. Romero et al., “Faa$T: A transparent auto-scaling cache for serverless applications,” in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 122–137.
[55] S. Shillaker and P. Pietzuch, “Faasm: Lightweight isolation for efficient stateful serverless computing,” in *Proc. 2020 USENIX Annu. Tech. Conf.*, 2020, pp. 419–433.

Huijuan Xiao received the master’s degree from the School of Computer Science, Fudan University. She is currently working towards the PhD degree with the School of Computer Science, Fudan University. Her research interests include database systems and cloud computing.

Shixi Yang received the bachelor’s degree from the Department of Atmospheric and Oceanic Science, Fudan University. He is currently working towards the master’s degree with the School of Computer Science, Fudan University. His research interests include database systems and high-performance computing.

Kai Zhang received the PhD degree from the University of Science and Technology of China in 2016. He is an associate professor with the School of Computer Science, Fudan University. He was a research fellow with the National University of Singapore and a visiting scholar with The Ohio State University. His research interests are mainly in the fields of parallel and distributed computing and database systems.

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.

Yinan Jing (Member, IEEE) received the PhD degree in computer science from Fudan University, Shanghai, China, in 2007. He is an associate professor with the School of Computer Science at Fudan University. He was also a visiting scholar with the Department of Computer Science, the University of Southern California. His current research interests include Big Data analytics, database systems, spatial and temporal data management, and data security and privacy. He is a member of ACM.

Zhenying He received the BS, MS, and PhD degrees in computer science from the Harbin Institute of Technology, China, in 1998, 2000, and 2006, respectively. Currently, he is an associate professor with the School of Computer Science, Fudan University. His current research interests include keywords search on structured data, query processing on RDFdata, and Big Data.

X. Sean Wang (Senior Member, IEEE) received the PhD degree in computer science from the University of Southern California. He is a distinguished professor with the School of Computer Science, Fudan University, Shanghai, China. Before joining Fudan University in 2011, he was the Dorothean chair professor with the University of Vermont. His research interests include data systems and data security. He is the fellow of CCF and the member of ACM.

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on January 27,2026 at 00:30:28 UTC from IEEE Xplore. Restrictions apply.