

DITTO: Online Memory Rightsizing and Value-Aware Eviction for Reducing Cold Starts in Serverless Platforms

Dongjae Lee*, Kyuli Park*, Yeonwoo Jeong, Hongsu Byun†, Sungyong Park†

Sogang University, Seoul, Republic of Korea

{raradong, kyuripark, akssus12, byhs, parksy}@sogang.ac.kr

Abstract—Modern serverless platforms create function instances based on user-specified memory limits, which often result in memory over-provisioning even when the average runtime footprint is small. This over-provisioning reduces warm container density under a fixed memory budget, leading to early pool saturation, frequent evictions, and increased cold-start latency. To address this fundamental limitation, we propose DITTO, a system that integrates online memory rightsizing with value-aware container management. DITTO dynamically adjusts container memory to match actual usage, reclaiming unused memory while safely avoiding out-of-memory (OOM) conditions. In addition, it prioritizes containers based on their benefit per unit memory using a score-based eviction and keep-alive policy. We implement DITTO in Apache OpenWhisk and evaluate it across diverse workloads. DITTO reduces cold-start rate by up to 25.7% and latency by up to 33.3% compared to vanilla OpenWhisk, and by up to 14.8% and 19.4%, respectively, compared to RainbowCake.

Index Terms—Serverless Computing, Container Caching, Coldstarts, Memory Rightsizing, Resource Management

I. INTRODUCTION

Serverless computing is a cloud-native execution model that allows developers to deploy event-driven functions without provisioning or managing server infrastructure. Upon receiving an invocation (e.g., HTTP request, a scheduled trigger, or a batch job), the cloud provider instantiates a function container, executes the function, and reclaims the container when it becomes idle. This on-demand model is well suited to short-lived, bursty workloads with variable arrival rates [1]–[9] and is now a first-class service on major platforms such as AWS Lambda [10] and Google Cloud Functions [11].

A key performance bottleneck in this model is the *cold start*: when no reusable container exists for an incoming request, the platform must create a new container and initialize its runtime, incurring latency that can be several to tens of times longer than the function execution [12]. Cold starts fundamentally occur when no reusable warm container is available at request arrival because the platform cannot maintain sufficient warm containers under a constrained memory budget (§III).

Motivation. Most serverless platforms retain completed containers in a *warm pool* so that subsequent invocations of the same function can bypass initialization. As the warm pool grows, incoming requests are more likely to find reusable cached containers and avoid cold starts. However, the pool operates under a fixed memory budget. Once this budget is

exhausted, the platform must evict existing warm containers to admit new ones, and any later invocation of an evicted function incurs a cold start, adding latency to the critical path. A key factor that exacerbates this problem is *memory over-provisioning*. Since users cannot accurately predict runtime memory demand at deployment time, they often provision memory well above actual usage to avoid out-of-memory (OOM) failures [13]. These inflated static limits consume the warm-pool budget regardless of actual utilization, reducing the number of containers that can coexist in the pool [14]. As a result, mitigating cold starts reduces to two core challenges: (1) *maximizing warm container density within a fixed memory budget*, and (2) *retaining the containers whose eviction would incur the highest cold-start penalty*.

Limitations of Existing Approaches. Prior work on mitigating cold starts can be broadly categorized into two directions [15], [16]. One direction reduces initialization overhead through cross-function sharing of container image layers, thereby shortening image loading and setup time [17]–[19]. The other direction optimizes warm-container eviction by ranking containers based on the ratio of cold-start cost to execution time, so that containers with the higher cold-start cost are retained longer [20].

These approaches focus primarily on either reducing initialization overhead or improving eviction ordering, without addressing the memory footprint of containers [21], [22]. In particular, layer sharing reduces initialization delay but does not reduce the runtime memory reserved for each container, which remains fixed by user-specified limits. Likewise, improved eviction policies can optimize which containers are removed, but cannot increase the number of containers that can be retained under a fixed memory budget. As a result, over-provisioned memory limits continue to consume warm-pool capacity, reducing the number of coexisting warm containers and increasing evictions and cold starts.

However, addressing this root cause is challenging. Online memory rightsizing must continuously adapt to bursty and unpredictable workloads, where overly aggressive downsizing can trigger OOM-induced function failures, while conservative sizing wastes memory and limits warm container density. At the same time, eviction policies must make decisions under continuously changing memory footprints and heterogeneous cold-start costs, making it difficult to consistently preserve the most valuable containers. To address these challenges, we propose DITTO, a system for serverless platforms that

*These authors are first co-authors and have contributed equally.

†Corresponding authors : Sungyong Park and Hongsu Byun.

improves memory efficiency and reduces cold starts.

DITTO. Our design is based on two key ideas. First, DITTO increases warm-container density through *online memory right-sizing*, dynamically adjusting container memory to match actual usage. Instead of reserving statically over-provisioned memory, it continuously reclaims unused memory while safely avoiding OOM failures via memory pressure monitoring. As a result, each container occupies only the memory it actually needs, allowing more warm containers to be retained under a fixed budget. Second, DITTO minimizes the impact of unavoidable evictions through a *value-aware container management policy* based on the *DITTO Score*, which jointly captures cold-start sensitivity and memory footprint. By prioritizing containers that deliver higher benefit per unit memory, DITTO preserves those with the greatest effect on latency reduction. The core novelty of DITTO lies in tightly coupling dynamic memory reclamation with value-aware eviction, thereby translating reclaimed memory into higher warm-container density.

Contributions. Our main contributions are as follows:

- **Structural analysis of memory over-provisioning (§III):** We demonstrate that statically configured memory limits fundamentally constrain warm container density, which in turn increases cold-start frequency.
- **Online memory rightsizing (§V-A):** We design an adaptive memory resizing mechanism that converges to actual usage without incurring OOM, improving memory efficiency.
- **Value-aware container management (§V-F):** We propose a DITTO Score for latency-aware keep-alive decisions based on cold-start sensitivity and memory efficiency.
- **Evaluation (§VI):** We implement DITTO in Apache OpenWhisk [23], achieving up to 25.7% fewer cold starts and 33.3% lower end-to-end latency across diverse workloads.

II. BACKGROUND

A. Serverless Computing Execution Model

In serverless platforms, each worker maintains a pool of containers to execute function invocations. As illustrated in Figure 1, a container transitions between several states during request handling. When a new invocation arrives and no reusable container is available, the platform creates and initializes a new container, resulting in a cold start. Once initialization completes, the container becomes warm and can be reused for subsequent invocations, bypassing creation and initialization to substantially reduce latency. However, warm containers cannot be retained indefinitely. Each worker operates under a finite memory budget, and once the pool becomes full, it must evict existing warm containers to admit new ones. In practice, many platforms adopt least recently used (LRU) eviction [24]–[27] and configure a fixed keep-alive timeout. This makes eviction decisions based on recency, ignoring the heterogeneous cost of cold starts across functions. This mismatch between eviction decisions and cold-start cost leads to suboptimal resource utilization and increased latency.

B. Cold-Start Overhead

Not all warm containers provide equal value under a constrained memory budget [28]. To determine which containers

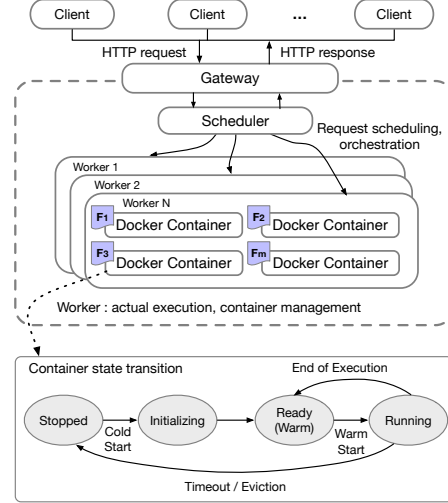


Fig. 1: Overview of serverless computing execution model.

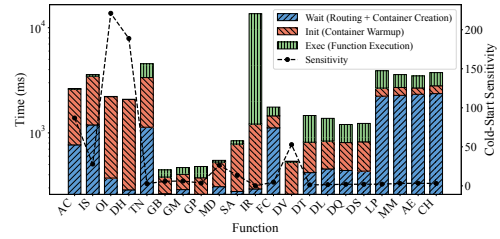


Fig. 2: Analysis of serverless function execution.

should be prioritized, it is necessary to quantify the benefit of retaining each container. Figure 2 shows the execution time breakdown of functions from open-source serverless benchmarks, including FunctionBench [29] and RainbowCake [17]. Detailed descriptions of each function and experimental setup are provided in §VI-A.

Each function consists of three stages: *Wait* (request handling and container creation), *Init* (dependency loading and runtime initialization), and *Exec* (actual function execution), whose proportions vary widely. We define *Cold Start Sensitivity* as the ratio of the sum of wait time and initialization time to actual execution time ($(T_{wait} + T_{init})/T_{exec}$). For example, SA has a sensitivity of 13.5, indicating over $13\times$ speedup from warm reuse, while IR (0.097) gains little due to dominant execution time. Other functions such as DH, DL, DQ, and DS also show low sensitivity, offering limited value in memory.

Since warm reuse benefits vary by function, eviction policies should prioritize high-sensitivity containers to maximize system-wide performance [16], [27], [30].

III. PROBLEM ANALYSIS

This section identifies two key factors that fundamentally increase cold-start frequency in pool-based serverless platforms.

P1. Static Memory Limits Lead to Increased Cold Starts. Users of serverless platforms cannot accurately predict runtime memory consumption, and configure memory limits well above actual usage to guard against OOM [13]. These inflated limits are reserved regardless of actual usage, reducing the

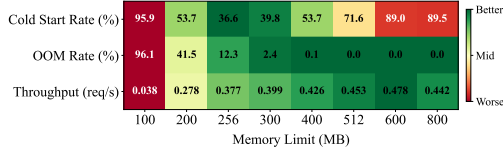


Fig. 3: Impact of static memory limits (100–800 MB) on cold-start rate, OOM rate, and throughput under a workload with an average memory footprint of approximately 256 MB.

number of warm containers that can coexist within a fixed memory budget. With fewer warm containers, the pool saturates quickly, forcing the platform to evict existing warm containers to admit new ones. Subsequent requests for evicted functions then incur cold starts.

Figure 3 shows how cold-start rate, OOM rate, and throughput vary as the memory limit ranges from 100 MB to 800 MB. Under a workload whose average memory footprint is approximately 256 MB, low limits trigger a sharp rise in OOM rate (96.1% at 100 MB), while high limits push cold-start rate up to 89.5%. While throughput generally improves with higher memory limits, this comes at the cost of increased cold starts. The 256–300 MB range offers the best trade-off between OOM and cold-start rates; however, this sweet spot is extremely narrow, making it difficult to choose a single static memory configuration that simultaneously avoids both OOM and excessive cold starts.

Figure 4 shows warm container count and cold-start frequency over time at 100 MB, 256 MB, and 400 MB. At 256 MB, the system maintains 20–30 warm containers, and cold starts remain consistently low. At 400 MB, the larger per-container footprint reduces warm containers to 17–20, leading to intermittent cold-start spikes due to more frequent pool saturation. At 100 MB, frequent OOM kills terminate containers, and ongoing requests may fail due to abrupt container termination. Each terminated container must be recreated for subsequent requests, resulting in bursts of cold starts.

These results highlight a fundamental limitation of static memory configuration: large limits trigger eviction-driven cold starts due to pool saturation, while small limits trigger OOM-driven cold starts due to container termination. No single static value can effectively avoid both. This calls for a dynamic mechanism that continuously adjusts container memory to match actual usage. However, increasing container density alone is insufficient. Even with more available capacity, the system must still decide which containers to retain when eviction is unavoidable.

P2. Sensitivity-Unaware Eviction Increases Cold Starts. Retaining warm containers is the primary strategy for reducing cold starts. However, when the pool is full and eviction is unavoidable, existing LRU-based policies select victims without considering the cold-start cost of each container.

As shown in Figure 2, cold-start sensitivity varies widely across functions. Consider two containers in a warm pool: SA (sensitivity 13.5) and IR (sensitivity 0.097). Keeping SA warm avoids substantial initialization overhead, whereas IR gains relatively little from reuse because its execution time

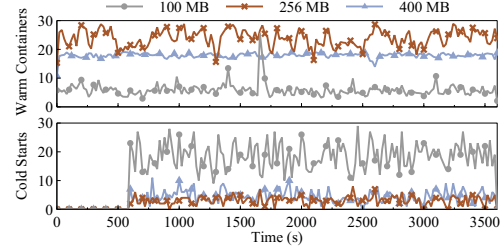


Fig. 4: Warm container count and cold-start frequency over time under 100 MB, 256 MB, and 400 MB configurations.

dominates. Yet, LRU makes decisions solely based on recency: if SA was last used 60 s ago and IR 59 s ago, LRU evicts SA first despite its much higher cold-start penalty. Since $(T_{wait} + T_{init}) = Sensitivity \times T_{exec}$, higher sensitivity directly implies higher cold-start penalty. Thus, such containers should be prioritized for warm starts.

The same problem arises with fixed keep-alive timeouts. Assigning an identical time-to-live (TTL) to all containers treats SA and IR equally, allowing high-sensitivity containers to expire prematurely. As a result, eviction decisions may trigger unnecessarily expensive cold starts, increasing overall latency. To minimize this impact under a constrained pool, eviction and keep-alive policies must explicitly account for cold-start sensitivity.

These observations reveal that reducing cold starts requires jointly addressing two orthogonal challenges: improving memory efficiency to increase warm container density, and making value-aware decisions to preserve high-impact containers. Existing approaches typically address only one of these aspects in isolation, limiting their effectiveness.

IV. DESIGN GOALS AND CHALLENGES

In this section, we present the design goals (G) of DITTO to address the two aforementioned limitations, along with their corresponding challenges (C).

A. Goals

G1. Online Memory Rightsizing. DITTO dynamically adjusts user-defined memory limits at runtime to match each container’s demand. It gradually reclaims unused memory through probing, increasing warm container density, and upsizes on throttling based on observed peak usage to prevent degradation. By continuously calibrating limits, DITTO adapts to workload variations without offline profiling.

G2. Value-Aware Container Management via DITTO Score. DITTO introduces the DITTO score, which incorporates cold-start sensitivity, memory footprint cost, and rightsizing status to quantify the retention value of each warm container. By integrating this score into keep-alive and eviction policies, the system prioritizes protecting containers that yield high utility per memory, thereby improving overall resource efficiency within a constrained pool.

B. Challenges

To achieve the aforementioned design goals, the following technical challenges must be addressed:

C1. Safe Rightsizing. The system must maximize memory reclamation during limit reduction without triggering OOM failures. If overly aggressive downsizing induces memory throttling, the limits must be swiftly restored to safeguard function execution latency.

C2. Workload-Adaptive Calibration. A function’s memory footprint can fluctuate depending on input data and invocation context. A memory limit confirmed to be safe in one invocation may trigger throttling in the next. Consequently, the system must dynamically adapt to both gradual shifts and sudden spikes online, without static profiling.

C3. Minimal Probing Overhead. The rightsizing mechanism must have negligible impact on function execution performance. Specifically, any throttling events triggered during the probing phase must be relieved instantaneously to minimize tail latency degradation.

C4. Node-Level Resource Coordination. Rightsizing decisions for individual containers must strictly align with the node’s physical memory budget. If the container pool is saturated with downsized instances and one suddenly requires upsizing, it can trigger a node-level OOM.

V. DESIGN OF DITTO

A. Architecture Overview

Components. As illustrated in Figure 5, DITTO comprises three core components: the Invoking Manager, the Rightsizing Agent, and the Container Pool.

- **Invoking Manager:** Manages function executions and container lifecycles. It maintains crucial metadata—such as container identifiers, states, and memory limits—to facilitate request routing and pool administration. Leveraging this metadata, DITTO computes a retention score for its novel eviction policy, which is designed to preserve the maximum number of high-value warm containers within the pool.
- **Rightsizing Agent:** Composed of Probe Manager and PSI Monitor, it is responsible for the memory rightsizing per container. The Probe Manager conducts the probing process to discover the optimal memory footprint, while the PSI Monitor instantaneously detects any throttling—triggered when actual usage exceeds the downsized limit during this phase—and immediately restores the boundary.
- **Container Pool:** Provides isolated runtime environments based on cgroup v2 [31].

To achieve safe downsizing without triggering OOM failures, DITTO leverages the following dynamic memory control mechanisms of cgroup: `memory.high`: Acts as a soft limit; when memory usage exceeds this boundary, the kernel throttles memory allocation instead of terminating the process. `memory.pressure`: Serves as an event trigger; if throttling occurs within a configured trigger window, it generates an asynchronous event.

Core Functions. DITTO continuously adjusts the memory limit of each container throughout its entire lifecycle to converge with its actual usage. This process comprises two operations and one synchronization mechanism:

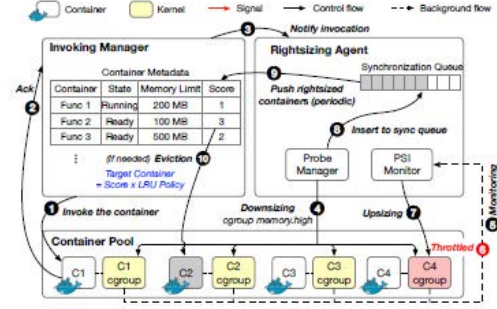


Fig. 5: Architecture of DITTO.

- **Downsizing:** Incrementally reduces `memory.high` to reclaim surplus memory, securing capacity to accommodate additional warm containers in the pool (§V-B).
- **Upsizing:** Restores `memory.high` based on observed peak usage upon detecting a throttling event, shielding function execution time from degrading by memory stalls (§V-C).
- **Synchronization:** Synchronizes the updated limit with the Invoking Manager upon every change, ensuring the pool manager maintains an accurate accounting of available capacity (§V-D).

Workflow. The end-to-end workflow of DITTO unfolds as follows. Upon receiving a function request, the Invoking Manager provisions a container ①. Once execution completes, the container returns an acknowledgment (Ack) with its identifier ②. The Invoking Manager then triggers the Rightsizing Agent to initiate or resume the probing phase ③, prompting the Probe Manager to incrementally downsize the memory boundary via `memory.high` ④. Concurrently, the PSI Monitor polls `memory.pressure` ⑤ and receives an asynchronous signal if throttling occurs ⑥. Upon detecting this stall, the PSI Monitor instantaneously reacts by upsizing the memory limit to mitigate the pressure ⑦. Following any capacity adjustment, the updated limit is enqueued into a Synchronization Queue ⑧. To minimize overhead, a background thread periodically flushes this queue in batches to the Invoking Manager, maintaining strict consistency of the global pool metadata ⑨. Finally, if the container pool reaches saturation and necessitates an eviction, the Invoking Manager isolates the optimal victim based on the DITTO Score ⑩.

B. Downsizing

Memory Step Size. To discover the safe memory lower bound for each container, the Probe Manager incrementally reduces `memory.high` by a specific step size. At each exploration epoch, the step size is computed and the memory limit is decreased only if at least one invocation has been processed since the previous epoch:

$$step = \frac{\alpha \cdot L_{cur}}{1 + Sensitivity}, \quad L_{cur} \leftarrow (L_{cur} - step) \quad (1)$$

where α is the baseline reduction rate governing the aggressiveness of exploration, L_{cur} is the current memory limit (`memory.high`), initialized to the user-specified upper limit M_{max} , and $Sensitivity = (T_{wait} + T_{init})/T_{exec}$ denotes

Algorithm 1 Online memory rightsizing

```

1: Input: user limit  $M_{max}$ , base rate  $\alpha$ , probe interval  $T_{probe}$ , sync threshold  $R_{min}$ 
2: State: current limit  $L$ , recorded peak  $P_{max}$ , consecutive throttle count  $n_{thr}$ , last
   synced limit  $L_{sync}$ 
3:  $L \leftarrow M_{max}$ ;  $P_{max} \leftarrow \text{memory.peak}$ ;  $n_{thr} \leftarrow 0$ ;  $L_{sync} \leftarrow M_{max}$ 
4: set  $\text{memory.high} \leftarrow L$ 
5: while container is active and  $n_{thr} < 3$  do
6:   wait  $T_{probe}$ 
7:   if no invocations in last interval then
8:     continue
9:   end if
10:  if PSI throttle detected then
11:    THROTTLERECOVERY ▷ §V-C
12:    continue
13:  end if
14:   $n_{thr} \leftarrow 0$  ▷ Reset consecutive count on successful probing
15:   $step \leftarrow \alpha \cdot L / (1 + \text{Sensitivity})$ 
16:   $L \leftarrow L - step$ 
17:  set  $\text{memory.high} \leftarrow L$ 
18:  update  $P_{max} \leftarrow \max(P_{max}, \text{memory.peak})$ 
19:  enqueue ( $container\_id$ ,  $L$ ) to Sync Queue
20: end while

```

the function's cold-start sensitivity. Equation 1 exhibits the following characteristics. First, as L_{cur} decreases, the step size shrinks proportionally, naturally shifting the exploration toward a more conservative approach over time. Second, functions with high cold-start sensitivity are assigned smaller step sizes, reflecting the severe cold-start penalty incurred in the event of throttling. Third, by relying exclusively on L_{cur} controlled by the system and Sensitivity tracked by the Invoking Manager, it completely bypasses the overhead of polling runtime memory usage.

Conditions for Suspension or Transition. The downsizing process is suspended or transitions under the following conditions: (1) upon throttling, it switches to the recovery procedure (§V-C); (2) after three consecutive throttling events, exploration is disabled; (3) when the step size falls below R_{min} for synchronization (§V-D), the agent enters an idle state. In this state, the container is monitored, and exploration resumes upon throttling after an upsize.

The peak memory record P_{max} is initialized by reading memory.peak upon the completion of the container's first execution, and is subsequently updated whenever a higher peak is observed. This value serves as the target limit during throttle recovery upscaling. The overall probing procedure is summarized in Algorithm 1.

C. Throttle Recovery and Upsizing

Upon detecting a throttling event, the PSI Monitor identifies that the memory limit is below demand, indicating overly aggressive exploration. DITTO then performs a staged response based on consecutive throttling events in the current epoch.

First Throttling Event. DITTO treats the first throttling as a transient spike and upsizes memory.high to P_{max} , the highest observed footprint. If the next execution succeeds, downsizing resumes from L_{cur} . If P_{max} is insufficient, throttling occurs again and escalates to the second stage.

Second Throttling Event. DITTO interprets repeated throttling as a genuine workload shift. Recognizing that the previous peak record no longer reflects the current memory demand, it upsizes memory.high to M_{max} to secure a sufficient safety margin. Subsequently, the downsizing process resumes

from the previous P_{max} . During this renewed exploration, P_{max} is naturally updated to capture the newly observed peak.

Third Consecutive Throttling Event. Three consecutive throttling events indicate a sustained high memory demand that can no longer be safely managed via exploration. Consequently, DITTO pins memory.high to M_{max} and permanently disables further downsizing exploration for the corresponding container.

Node-Level Safety. Before upsizing, the system checks the node's available memory. Since downsizing increases container density beyond the original M_{max} -based capacity, an unconditional upsize may exceed physical limits. If insufficient, DITTO aborts the upsizing and triggers an OOM event, ensuring node stability.

D. Synchronization

Synchronization. Synchronization is a background mechanism that keeps the Invoking Manager apprised of each container's active memory limit without terminating the ongoing exploration phase. Whenever a downsizing or upsizing operation modifies memory.high , the Probe Manager enqueues a tuple containing the container identifier and the updated limit, ($container_id$, L_{cur}), into the Synchronization Queue. A dedicated background thread periodically processes these queued updates in batches to synchronize the Invoking Manager. The conditions for inserting an update into the Synchronization Queue are as follows:

Minimum Reclamation Threshold. The cumulative amount of memory reclaimed since the last synchronization must meet or exceed the minimum reclamation threshold, R_{min} :

$$M_{last_delivered} - L_{cur} \geq R_{min} \quad (2)$$

where $M_{last_delivered}$ is the memory limit previously sent to the Invoking Manager. This prevents overloading the Invoking Manager with trivial updates and ensures each synchronization frees enough space for additional containers. It also serves as a convergence criterion: when the step size can no longer satisfy R_{min} , updates cease and the system enters an idle state.

Temporal Stability. Even if the R_{min} threshold is met, the system requires a minimum elapsed time, T_{stable} , after applying the current limit. This temporal safeguard prevents the premature dispatch of updates during transient periods of low memory utilization:

$$T_{stable} = \begin{cases} T_{init} & \text{if } t_{iat} = 0 \\ \min(\max(\alpha_t \cdot t_{iat}, T_{min}), T_{max}) & \text{otherwise} \end{cases} \quad (3)$$

where α_t controls the tradeoff between stability and responsiveness, and t_{iat} denotes Inter-Arrival Time (IAT) of function. Functions with short intervals enable early synchronization, while infrequent functions require longer observation.

Once an update satisfying both conditions is delivered, the Invoking Manager updates its pool metadata. To ensure consistency, the agent removes the entry from the Synchronization Queue only after receiving an acknowledgment (Ack). If throttling occurs during the stability period, pending updates are discarded and the system switches to throttle recovery.

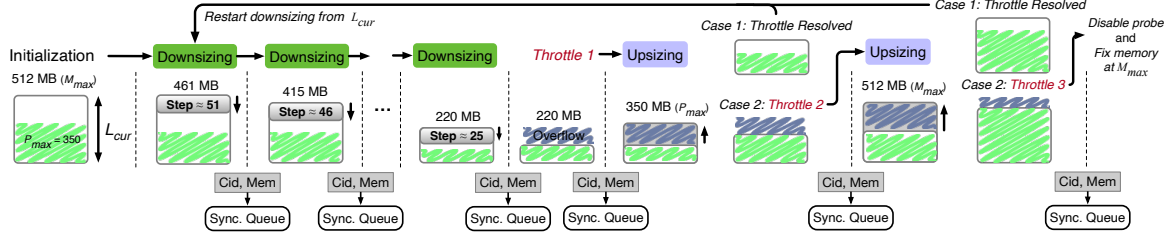


Fig. 6: Overall rightsizing scenario of DITTO ($M_{max} = 512$ MB, $P_{max} = 350$ MB, $\alpha = 0.2$, $Sensitivity = 1$).

E. Rightsizing Scenario

Figure 6 illustrates an end-to-end rightsizing scenario configured with $M_{max} = 512$ MB, $P_{max} = 350$ MB, $\alpha = 0.2$, and $Sensitivity = 1$.

Following the completion of the container's initial execution, the system initiates an incremental reduction of `memory.high` starting from M_{max} . At each step, the reduction size is governed by $step = \alpha \cdot L_{cur} / (1 + Sensitivity)$, ensuring that the step size decreases proportionally as L_{cur} shrinks. For example, Step 1 reduces the limit by approximately 51 MB (from 512 to 461), and Step 2 reduces it by roughly 46 MB (down to 415). By Step 8, the step size diminishes to around 25 MB, bringing L_{cur} to 220 MB. At every step, the container identifier and the updated memory limit are inserted into the Synchronization Queue.

The subsequent section demonstrates the recovery and up-sizing process in the event of throttling. When the actual memory usage exceeds the limit at $L_{cur} = 220$ MB, a memory stall occurs. The PSI Monitor detects this stall and initiates the first-stage throttle recovery, up-sizing `memory.high` to P_{max} (350 MB). If the subsequent function execution succeeds, the system resumes the downsizing process starting from $L_{cur} = 220$ MB, the exact point where the throttle occurred.

However, if a memory stall occurs even at P_{max} , the state escalates to a second throttling event. The system upsizes `memory.high` to M_{max} (512 MB) to secure a sufficient safety margin, and then restarts the downsizing from P_{max} (350 MB). If another memory stall happens during this phase, it is classified as a third consecutive throttle. Consequently, the system permanently disables the exploration and pins `memory.high` to M_{max} . Thereafter, the container safely serves requests using the user-defined memory limit.

F. DITTO Score based Eviction Policy

As shown in §III, cold-start costs vary significantly across different functions. Therefore, choosing which container to remove during eviction impacts the overall cold starts. To address this, DITTO defines V_{DITTO} to quantify the retention value of warm containers, directly incorporating this metric into its eviction and keep-alive policies.

DITTO Score Calculation. V_{DITTO} quantifies the benefit and cost associated with retaining a container as follows:

$$V_{DITTO}(c) = \frac{Sensitivity(c)}{M_{target}(c)} \times \alpha_{status}(c) \quad (4)$$

- **Cold-Start Sensitivity** ($Sensitivity = (T_{wait} + T_{init})/T_{exec}$). This ratio represents the sum of wait and initialization time relative to the execution time. A higher value indicates greater time savings upon warm reuse. For instance, an ML inference function loading heavy libraries receives a higher score than a lightweight computational function.
- **Memory Footprint Cost** ($1/M_{target}$). M_{target} denotes the memory currently occupied by the container. Given equal utility, preserving containers with smaller footprints is advantageous for increasing the packing density of warm containers in the pool. Through rightsizing, M_{target} explicitly reflects the dynamically downsized limit rather than the user-defined configuration.
- **Rightsizing Incentive** (α_{status}). This prioritizes the retention of successfully downsized containers, preventing the overhead of re-exploration that would occur if they were evicted and subsequently recreated.

$$\alpha_{status}(c) = \begin{cases} \delta & (0 < \delta < 1) \text{ if disabled} \\ 1.0 & \text{if probing} \\ 1 + \frac{M_{max} - M_{target}(c)}{M_{max}} & \text{if synchronized} \end{cases} \quad (5)$$

Containers that complete synchronization receive a weight proportional to reclaimed memory. Containers with disabled exploration due to repeated throttling are assigned a penalty $\delta < 1$, while those under exploration receive a weight of 1.0. **Eviction and Keep-alive.** When the container pool is saturated and an eviction is triggered, DITTO selects a victim based on an Eviction Score, which combines the V_{DITTO} metric with the container's idle time:

$$EvictionScore(c) = V_{DITTO}(c) \times \frac{1}{t_{idle}(c)} \quad (6)$$

Containers with the lowest Eviction Scores are prioritized for eviction. While a simple LRU policy relies solely on recent usage—risking the eviction of containers with high cold-start cost—DITTO jointly evaluates retention utility and idle time to safeguard containers that incur severe cold-start penalty. Keep-alive durations are dynamically assigned in proportion to the container's retention value:

$$T_{keep}(c) = T_{base} + \ln(1 + V_{DITTO}(c)) \quad (7)$$

Additional time proportional to V_{DITTO} is added to T_{base} , with a logarithmic term to prevent indefinite occupation. This prioritizes containers with high cold-start cost, small memory footprint, and completed rightsizing.

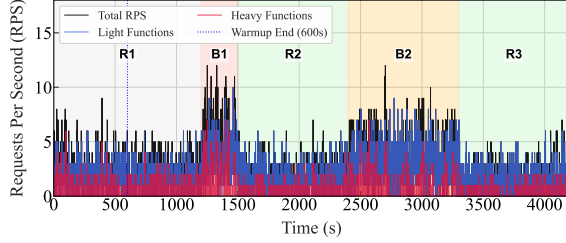


Fig. 7: Dynamic Workload. Total RPS is the sum of Light functions and Heavy functions. RN and BN represent regular and burst phases, respectively.

VI. EVALUATION

This section presents a comprehensive evaluation of DITTO across real-world workloads.

A. Experiment Setup

Implementation. DITTO is implemented as a Go agent co-located with the Invoker, a core component of Apache OpenWhisk. DITTO operates as a daemon on each worker node, leveraging cgroup v2 [31] and the cgroupfs driver to directly manipulate Linux cgroup v2 interfaces and kernel PSI. We modified the Akka FSMs within the OpenWhisk Invoker module—specifically, the ContainerPool and ContainerProxy, which are responsible for container pool management and action execution, respectively.

Platform. Our experiments were conducted on clusters, which consist of one master node and two worker nodes. Master node has Intel(R) Core(TM) i5-7500 3.40GHz CPU and 16 GB DDR4 DRAM, which runs client process. Each worker node has 13th Gen Intel(R) Core(TM) i5-13600K 5.1GHz and 32 GB DDR4 DRAM. In work node, we set container pool size as 15 GB, adding up to 30 GB. In addition, we disable hyperthreading and use the hashing policy provided in OpenWhisk for load balancing between worker nodes.

Workloads. The workloads used in our experiments are shown in Table I. We use a subset of functions from the open-source serverless benchmark Functionbench [29] and RainbowCake [17]. For payloads, we profile execution time for each input and classify functions into light (up to 200 ms) and

heavy (up to 10 s). Unless otherwise specified, payloads follow a Zipf distribution with maximum execution times of 200 ms and 10 s for each type.

Traffic. We use the Azure Function Trace [13] and assign different Inter-arrival Time (IAT) and Coefficient of Variation (CV) to light and heavy functions. Light functions have $IAT \leq 0.4$ and $CV \leq 1$, while heavy functions have $IAT \geq 1$ and $CV \geq 2$, generating intermittent bursts that increase load and trigger container eviction. We combine these functions to construct two workloads. In the static workload, functions have fixed IAT and CV throughout the experiment. In the dynamic workload, functions have phase-varying IAT and CV within their respective ranges (Figure 7).

Comparison Targets. We selected the following pool-based open source serverless platforms to compare DITTO.

- **Vanilla:** Default Apache OpenWhisk.
- **RainbowCake:** Container partial sharing + Dynamic keep-alive duration.
- **DITTO:** Container downsizing agent + New eviction policy + Dynamic keep-alive duration.

The keep-alive duration can affect platform performance. While OpenWhisk relies on a static keep-alive timeout, DITTO and RainbowCake employ a dynamic keep-alive strategy. We configure OpenWhisk with its default keep-alive duration of 10 minutes. The value for α , used for DITTO step size calculation V-B, was set to 0.4 throughout the experiment.

B. Overall Performance

Figure 8 presents the throughput and average latency, and shows the tail-latency distribution as a CDF under the Static and Dynamic workloads.

Static Workload. Figure 8(a) shows the throughput and average latency. DITTO achieves up to 26% higher throughput and 33.3% lower latency than Vanilla, and 19.4% lower latency than RainbowCake. This improvement stems from DITTO’s rightsizing mechanism, which increases warm container density in the pool and thereby raises the likelihood of warm-container reuse upon request arrival. In contrast, RainbowCake improves throughput and latency over Vanilla through container image layer sharing, but it does not reduce the runtime memory footprint of each container. As a result, it remains less effective than DITTO.

In Figure 8(b), DITTO consistently appears the furthest to the left, indicating the lowest latency across the entire distribution. At P95 in particular, DITTO reduces latency by 49.9% compared with Vanilla and by 21.7% compared with RainbowCake. This result indicates that DITTO mitigates tail latency by reducing cold-start frequency.

Dynamic Workload. Figure 8(c) shows that all three systems achieve comparable throughput, as the aggregate request rate remains within the service capacity of each system. In contrast, their latency differs substantially. DITTO reduces average latency by 22.9% and 10.9% compared with Vanilla and RainbowCake, respectively. At P50, DITTO further lowers latency by 66.9% relative to Vanilla and by 60.9% relative to RainbowCake, indicating a clear improvement in median response time.

TABLE I: Target Benchmark Workloads.

Language	Function	Domain
Node.js	Auto Complete (AC)	Web App
	Image Sizing (IS)	Multimedia
	Optical Character Recognition (OI)	Machine Learning
	Dynamic HTML (DH)	Web App
	Thumbnail Generator (TN)	Multimedia
Python	Graph BFS (GB)	Scientific Computing
	Graph MST (GM)	Scientific Computing
	Graph Pagerank (GP)	Scientific Computing
	Markdown (MD)	Web App
	Sentiment Analysis (SA)	Machine Learning
	Image Recognition (IR)	Machine Learning
	File Compression (FC)	Scientific Computing
	DNA Visualization (DV)	Scientific Computing
	Linpack (LP)	Scientific Computing
	Matrix Multiplication (MM)	Scientific Computing
	AES Encryption (AE)	Security
	Chameleon (CH)	Web App
	RNN Generation (RG)	Machine Learning
	Video Face Detection (VF)	Multimedia
Java	Data Transform (DT)	Data Analysis
	Data Load (DL)	Data Analysis
	Data Query (DQ)	Data Analysis
	Data Statistics (DS)	Data Analysis

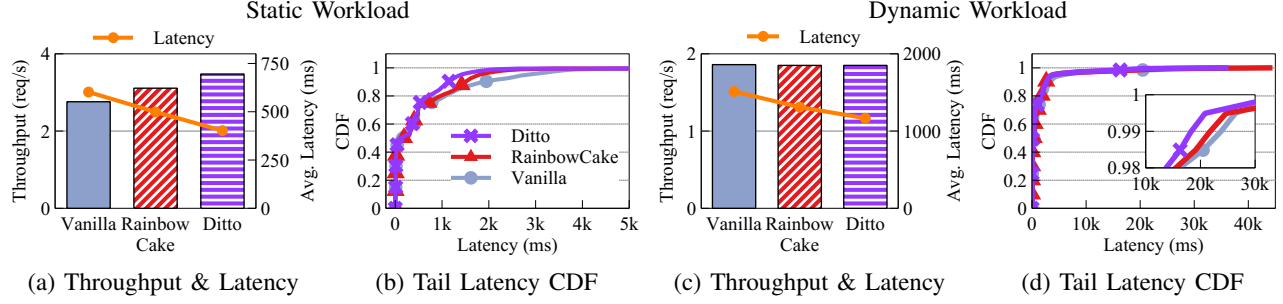
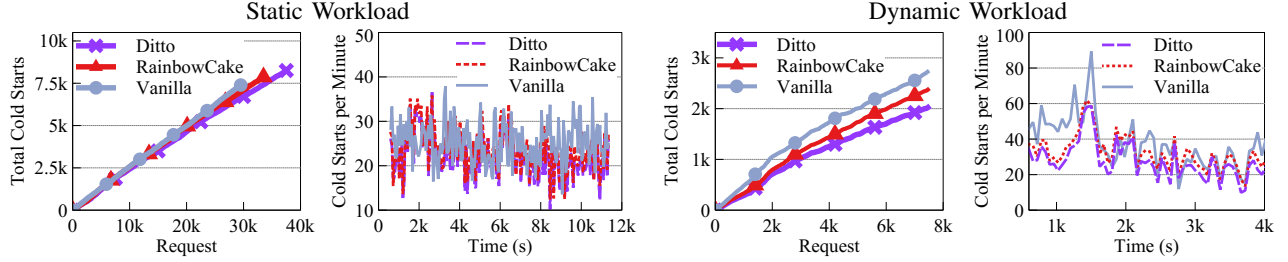


Fig. 8: Overall performance comparison. (a)–(b) and (c)–(d) are results of static workload and dynamic workload, respectively.



(a) Cumulative sum of cold starts (b) Cold starts per min (c) Cumulative sum of cold starts (d) Cold starts per min

Fig. 9: Cold-start overhead. Cumulative count per requests (a) and (c), and per-minute count (b) and (d). (a)–(b) and (c)–(d) are results of static workload and dynamic workload, respectively.

The CDF in Figure 8(d) shows that DITTO remains leftmost across the full range. Despite increased tail latency under bursty dynamic workloads, DITTO consistently achieves the lowest tail latency, reducing it by 23.2% compared to Vanilla and 15.6% compared to RainbowCake, demonstrating stable performance under fluctuating load.

C. Cold Start Overhead

Figure 9 shows the cumulative number of cold starts and the per-minute cold-start frequency under the Static and Dynamic workloads.

Static Workload. In Figure 9(a), DITTO records the largest cumulative number of cold starts in absolute terms by the end of the experiment. However, DITTO achieves the lowest cold-start ratio per request at 22.0%, compared with 25.1% for Vanilla and 23.5% for RainbowCake. This indicates that DITTO incurs fewer cold starts for the same amount of work.

As shown in Figure 9(b), Vanilla exhibits high variance with intermittent spikes, whereas DITTO maintains the lowest and most stable cold-start frequency. This improvement comes from rightsizing, which increases warm container density and reduces evictions. RainbowCake improves over Vanilla, but remains less effective than DITTO because it does not reduce the runtime memory footprint of each container.

Dynamic Workload. Figure 9(c) shows that DITTO reduces cumulative cold starts by 25.7% compared with Vanilla and by 14.8% compared with RainbowCake. Its curve also becomes more gradual over time, indicating that rightsizing progressively improves warm container density.

In Figure 9(d), DITTO consistently maintains the lowest per-minute cold-start frequency, and the gap widens in the later

phase of the experiment. This trend indicates that rightsizing gradually expands effective pool capacity by optimizing the memory footprint of more containers over time.

D. Cold Start per Function

Figure 10 shows per-function serving latency and cold-start rate. Descriptions of each function are provided in Table I. DITTO consistently reduces both metrics across most functions. The largest gains appear in functions with high initialization cost such as LP and MM, where DITTO cuts serving latency by up to 65% over Vanilla. Functions with small memory footprints such as FC and OI see their cold-start rates drop below 1%, as rightsizing frees sufficient pool space to keep these lightweight containers warm almost indefinitely.

E. Ablation Study

1) *Analysis of α :* To evaluate the impact of the base step rate α in Equation 1, we varied α from 0.2 to 0.8 under the same workload. Table II reports the cold-start rate and average latency for each setting.

A small α makes downsizing conservative, slowing convergence and delaying memory reclamation. As a result, the pool cannot increase the number of warm containers quickly enough, leading to a higher cold-start rate. This is evident at $\alpha = 0.2$, which shows the highest cold-start rate. In contrast, a large α makes downsizing too aggressive,

TABLE II: Impact of base reduction rate α on cold-start rate and average latency.

α	CS Rate (%)	Avg Lat. (ms)
0.2	30.50	1521.4
0.4	25.00	1529.0
0.5	27.20	1580.5
0.6	27.80	1619.9
0.8	28.30	1600.3

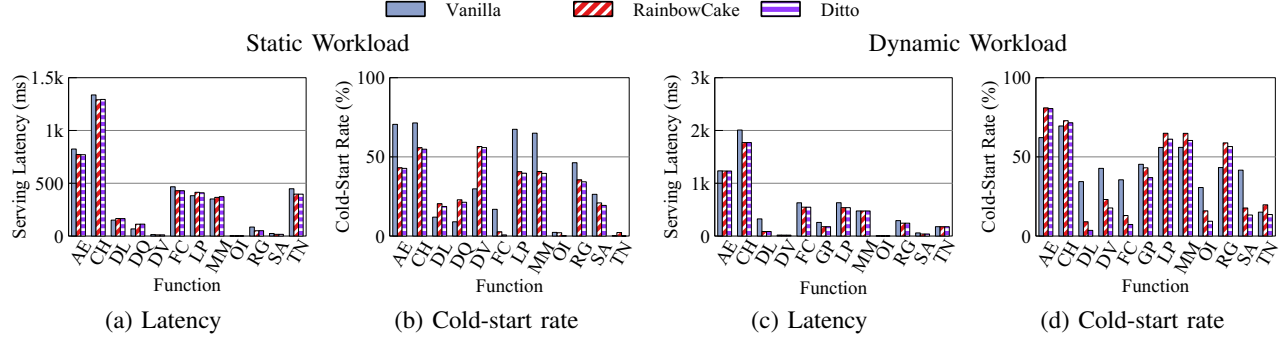


Fig. 10: Correlation between average latency and frequency of cold start per serverless function. (a)–(b) and (c)–(d) are results from static and dynamic workloads, respectively.

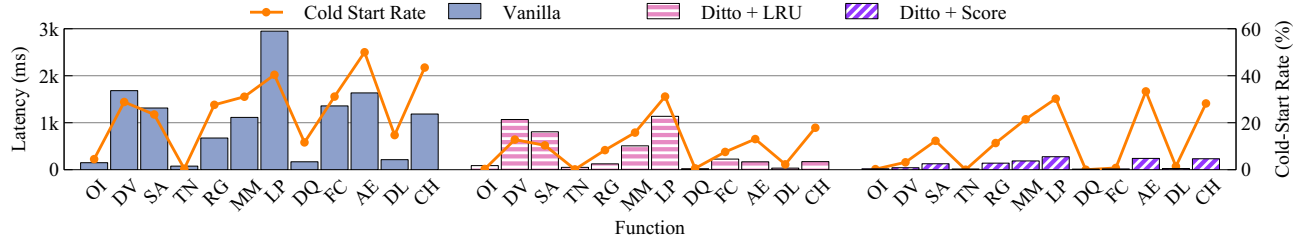


Fig. 11: Cold-start rate and latency comparison for downsizing and eviction policy. Left y -axis: serving latency (ms); right y -axis: cold-start rate (%). Functions on the x -axis are ordered by decreasing cold-start sensitivity.

causing frequent throttling and repeated upsizing–downsizing transitions that increase both cold starts and latency. Overall, $\alpha = 0.4$ achieves the lowest cold-start rate while keeping latency low, indicating the best balance between convergence speed and throttling overhead. We therefore use $\alpha = 0.4$ in the rest of the experiments.

2) *Eviction Ablation*: To isolate the contribution of each mechanism, we compare three configurations: Vanilla, DITTO+LRU (rightsizing only, default LRU eviction), and DITTO+Score (rightsizing + Score-based eviction). Figure 11 shows per-function cold-start latency and cold-start rate, ordered by decreasing sensitivity.

Effect of Rightsizing. Rightsizing alone reduces the overall cold-start rate from 19.6% to 6.4%, a 67.3% reduction over Vanilla. Cold-start latency also drops across most functions, with the largest gains in LP, DV, and AE. The gains result from higher warm container density: rightsizing reclaims unused memory, allowing the pool to retain more warm containers.

Effect of DITTO Score based Eviction. Adding DITTO score based eviction selectively reduces cold starts for high-sensitivity functions. DV (sensitivity 29.8) drops from 12.81% to 3.15% cold-start rate and from 1068 ms to 42 ms in cold-start latency. FC drops from 7.56% to 0.75%.

In contrast, low-sensitivity functions such as AE (0.68) and CH (0.40) see their cold-start rates rise from 13.04% to 33.33% and 17.86% to 28.21%, respectively. This is by design: the score policy protects high-sensitivity containers from eviction at the expense of low-sensitivity ones. Because AE and CH have short initialization times relative to execution, the latency impact of their additional cold starts is small. AE’s cold-start latency increases only from 168 ms to 241 ms, while the

elimination of cold starts for DV and FC yields a large net reduction in system-wide latency.

Overall Performance. Figure 12 shows that DITTO+LRU improves throughput by 15.2% and reduces latency by 48.9% over Vanilla, while DITTO+Score further improves throughput by 21.7% and reduces latency by 63.1%. This is achieved by selectively reducing cold starts for high-sensitivity functions, which dominate overall latency.

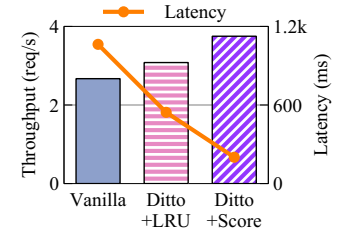


Fig. 12: Ablation: throughput and average latency.

F. OOM Analysis

To understand how static memory limits affect OOM, cold starts, and latency, we run the same workload for 30 minutes under four static configurations (400 MB, 500 MB, 600 MB, 1 GB) and under DITTO starting from $M_{max} = 1$ GB. The workload follows a Zipf distribution [32] with variable per-invocation input sizes, producing fluctuating memory demand. Table III shows the results.

Low Memory Limits: OOM and Request Loss. The 400 MB and 500 MB settings yield the lowest cold-start rates (4.95% and 12.21%), but incur 58 and 46 OOM events respectively. Occasional bursty inputs exceed the static limit, and each OOM kill terminates the container immediately, losing the in-flight request without retry. Because these lost requests never complete, they are excluded from cold-start statistics, making the reported cold-start rates appear deceptively low.

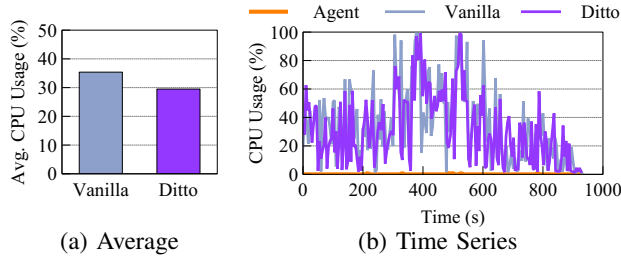


Fig. 13: CPU Overhead. (a) Average CPU usage per node. (b) Node CPU usage in time series, along with agent CPU usage.

High Memory Limits: Increased Cold Starts and Latency. The 600 MB and 1 GB settings eliminate OOM entirely, but the larger per-container footprint raises cold-start rates to 19.67% and 38.63%. The 1 GB setting shares the same M_{max} as DITTO, yet without rightsizing every container reserves its full 1 GB, resulting in the worst average latency (1285.7 ms) and P99 (4150.4 ms) among all configurations. The 600 MB setting appears to be the best static trade-off, with zero OOM and a moderate cold-start rate. However, this balance point depends on the current workload’s memory distribution and cannot be determined at deployment time.

DITTO: Lowest Latency with Zero OOM. DITTO starts at 1 GB to prevent OOM and rightsizes each container down to its actual footprint. Compared to the 1 GB static baseline, it reduces the cold-start rate from 38.63% to 26.88% and average latency from 1285.7 ms to 663.3 ms (−48.4%). Compared to the best static setting (600 MB), DITTO has a higher cold-start rate but delivers 25.1% lower average latency and 39.9% lower P99 latency. Rightsizing reclaims unused memory and raises warm container density in the pool, increasing the probability that incoming requests find a warm container. DITTO therefore requires only a safe M_{max} from the user and adapts automatically to the workload, delivering the lowest latency among all OOM-free configurations.

G. Overhead Analysis

Figure 13 compares the CPU utilization between Vanilla OpenWhisk and DITTO. As shown in Figure 13(a), DITTO consumes less average CPU than Vanilla despite running background rightsizing agents. The agent itself introduces negligible overhead (Figure 13(b)); the net reduction stems from avoiding repeated container creation and destruction, which causes Vanilla’s CPU spikes at 300s and 600s.

VII. RELATED WORK

Recent studies have explored various strategies to reduce cold starts in pool-based serverless platforms by improving warm container retention under constrained node memory. These approaches can be broadly classified into two categories.

TABLE III: Impact of static memory configuration.

Config	OOM	CS Rate (%)	Avg Lat. (ms)	P99 (ms)
400MB	58	4.95	587.0	3510.8
500MB	46	12.21	746.8	3674.1
600MB	0	19.67	886.1	3775.9
1GB	0	38.63	1285.7	4150.4
DITTO	0	26.88	663.3	2267.6

Warm Container Provisioning. One line of work predicts invocation timing and pre-warms containers before requests arrive, or shares container components to increase packing density within a fixed memory budget. Kraken [33] and Cypress [34] analyze inter-function dependencies and resource requirements to launch containers proactively, enabling warm starts for subsequent invocations. RainbowCake [17] and Pagurus [22] share runtime layers across functions, reducing initialization overhead. Parrotfish [35] and input-aware sizing methods [36]–[38] reduce per-container memory through offline profiling. However, prediction-based provisioning is vulnerable to misprediction under bursty traffic, which can waste memory or trigger OOM. Layer-sharing techniques [39], [40] are tied to specific runtime structures, may not accelerate warm starts at every sharing granularity, and introduce communication overhead for coordination.

Container Eviction Policies. Another line of work focuses on which warm containers to evict when node memory is exhausted. These studies argue that traditional eviction policies such as LRU are ill-suited to serverless workloads and propose differentiated container valuation based on various metrics. FaaSCache [41] combines container size and invocation frequency so that frequently called, small-footprint functions remain in the pool longer. S-Cache [42] caches containers for serverless edge computing based on container frequency, size, and cold-start time. CIDRE [20] adopts a concurrency-aware strategy that preferentially retains idle containers of bursty functions to raise their warm start probability. However, none of these approaches reduce the likelihood of eviction itself, because the pool capacity remains constrained by statically over-provisioned memory limits.

VIII. DISCUSSION

DITTO assumes independent memory and CPU control (cgroup v2). On platforms that couple CPU shares to memory limits (e.g., AWS Lambda), rightsizing would also reduce CPU and may harm CPU-bound functions, so adapting DITTO to such settings is left as future work. For functions with strict tail latency SLOs, repeated throttling and recovery cycles may degrade responsiveness, and DITTO therefore supports a per-function opt-out flag that pins the limit to M_{max} .

IX. CONCLUSION

Memory over-provisioning accelerates warm-pool saturation, triggers frequent evictions, and ultimately increases cold-start frequency. To address this issue, our proposed DITTO devises a safe online memory downsizing and score-based eviction policy to reduce cold start latency. Our extensive evaluation across diverse real-world workloads demonstrates that DITTO reduces end-to-end latency by up to 33.3% compared to vanilla OpenWhisk and by 19.4% compared to RainbowCake.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2024-00453929) (RS-2024-00416666).

REFERENCES

- [1] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pp. 1288–1296, 2019.
- [2] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 257–262, 2018.
- [3] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: a serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC '19*, (New York, NY, USA), p. 13–24, Association for Computing Machinery, 2019.
- [4] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC '18*, (New York, NY, USA), p. 263–274, Association for Computing Machinery, 2018.
- [5] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalariao, A. Sivaraman, G. Porter, and K. Winstead, "Encoding, fast and slow: low-latency video processing using thousands of tiny threads," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation, NSDI'17*, (USA), p. 363–376, USENIX Association, 2017.
- [6] J. Enes, R. R. Expósito, and J. Touriño, "Real-time resource scaling platform for big data workloads on serverless environments," *Future Generation Computer Systems*, vol. 105, pp. 361–379, 2020.
- [7] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17*, (New York, NY, USA), p. 445–451, Association for Computing Machinery, 2017.
- [8] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden, "Starling: A scalable query engine on cloud functions," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20*, (New York, NY, USA), p. 131–141, Association for Computing Machinery, 2020.
- [9] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pp. 1063–1075, 2019.
- [10] Amazon, "Aws lambda." [Online]. Available: <https://aws.amazon.com/lambda/>. Accessed: Dec. 8, 2025.
- [11] Google, "Cloud functions." [Online]. Available: <https://cloud.google.com/serverless>. Accessed: Dec. 8, 2025.
- [12] X. Cai, Q. Sang, C. Hu, Y. Gong, K. Suo, X. Zhou, and D. Cheng, "Incendio: Priority-based scheduling for alleviating cold start in serverless computing," *IEEE Transactions on Computers*, vol. 73, no. 7, pp. 1780–1794, 2024.
- [13] M. Shahrad, R. Fonseca, I. n. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider," *USENIX ATC'20*, (USA), USENIX Association, 2020.
- [14] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 127–144, 2014.
- [15] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: skip redundant paths to make serverless fast," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–15, 2020.
- [16] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proceedings of the 21st international middleware conference*, pp. 1–13, 2020.
- [17] H. Yu, R. Basu Roy, C. Fontenot, D. Tiwari, J. Li, H. Zhang, H. Wang, and S.-J. Park, "Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 335–350, 2024.
- [18] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, (Boston, MA), pp. 923–935, USENIX Association, July 2018.
- [19] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 300–312, 2018.
- [20] Q. Liu, Y. Cheng, H. Shen, A. Wang, and B. Balaji, "Concurrency-informed orchestration for serverless functions," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 147–161, 2025.
- [21] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless era," in *Proceedings of the 21st International Middleware Conference*, pp. 280–295, 2020.
- [22] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li, et al., "Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing," in *2022 USENIX annual technical conference (USENIX ATC 22)*, pp. 69–84, 2022.
- [23] Apache, "Openwhisk." [Online]. Available: <https://openwhisk.apache.org/>. Accessed: Dec. 8, 2025.
- [24] A. C. Zhou, R. Huang, Z. Ke, Y. Li, Y. Wang, and R. Mao, "Tackling cold start in serverless computing with multi-level container reuse," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 89–99, 2024.
- [25] B. Sethi, S. K. Addya, and S. K. Ghosh, "Lcs : Alleviating total cold start latency in serverless applications with lru warm container approach," *Proceedings of the 24th International Conference on Distributed Computing and Networking*, 2023.
- [26] N. Buitrago, H. Camacho, M. Jimeno, C. Viloria-Núñez, J. A. Cardona, and A. Salazar, "Parsec: An adaptive and efficient platform for reducing cold start in serverless computing," *IEEE Transactions on Services Computing*, vol. 18, no. 6, pp. 4082–4095, 2025.
- [27] C. Qiao, C. Wang, Z. Zhang, Y. Ji, and X. Gao, "Caching aided multi-tenant serverless computing," *arXiv preprint arXiv:2408.00957*, 2024.
- [28] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling quality-of-service in serverless computing," in *Proceedings of the 11th ACM symposium on cloud computing*, pp. 311–327, 2020.
- [29] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 502–504, IEEE, 2019.
- [30] C. Lee, Z. Zhu, T. Yang, Y. Huo, Y. Su, P. He, and M. R. Lyu, "Spes: towards optimizing performance-resource trade-off for serverless functions," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pp. 165–178, IEEE, 2024.
- [31] Red Hat, "World domination with cgroups in rhel 8: wel- come cgroups v2!," <https://www.redhat.com/en/blog/world-domination-cgroups-rhel-8-welcome-cgroups-v2>, 2019. Accessed: Dec. 8, 2025.
- [32] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, vol. 1, pp. 126–134, IEEE, 1999.
- [33] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 153–167, 2021.
- [34] V. M. Bhasi, J. R. Gunasekaran, A. Sharma, M. T. Kandemir, and C. Das, "Cypress: Input size-sensitive container provisioning and request scheduling for serverless platforms," in *Proceedings of the 13th Symposium on Cloud Computing*, pp. 257–272, 2022.
- [35] A. Moghimi, J. Hattori, A. Li, M. Ben Chikha, and M. Shahrad, "Parrotfish: Parametric regression for optimizing serverless functions," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pp. 177–192, 2023.
- [36] R. Cordingley, S. Xu, and W. Lloyd, "Function memory optimization for heterogeneous serverless platforms with cpu time accounting," in *2022 IEEE international conference on cloud engineering (IC2E)*, 2022.
- [37] S. Agarwal, M. A. Rodriguez, and R. Buyya, "Input-based ensemble-learning method for dynamic memory configuration of serverless computing functions," in *2024 IEEE/ACM 17th International Conference on Utility and Cloud Computing (UCC)*, pp. 346–355, IEEE, 2024.
- [38] G. Safaryan, A. Jindal, M. Chadha, and M. Gerdnt, "Slam: Slo-aware memory optimization for serverless applications," in *2022 IEEE 15th international conference on cloud computing (CLOUD)*, pp. 30–39, IEEE, 2022.
- [39] L. Gu, Z. Chen, H. Xu, D. Zeng, B. Li, and H. Jin, "Layer-aware collaborative microservice deployment toward maximal edge throughput," in *IEEE INFOCOM 2022-IEEE Conference on computer communications*, pp. 71–79, IEEE, 2022.
- [40] D. Zeng, H. Geng, L. Gu, and Z. Li, "Layered structure aware dependent microservice placement toward cost efficient edge clouds," in *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pp. 1–9, 2023.
- [41] A. Fuerst and P. Sharma, "Faas-cache: keeping serverless computing alive with greedy-dual caching," in *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*, pp. 386–400, 2021.
- [42] C. Chen, L. Nagel, L. Cui, and F. P. Tso, "S-cache: Function caching for serverless edge computing," in *Proceedings of the 6th international workshop on edge systems, analytics and networking*, pp. 1–6, 2023.