

Unipool: Efficient Management of Pre-Provisioned Resources for Container Platforms

Hanfei Yu*
Stevens Institute of Technology
Hoboken, NJ, United States
hyu42@stevens.edu

Enrique Saurez
Microsoft Azure Research – Systems
Redmond, WA, United States
esaurez@microsoft.com

Íñigo Goiri
Microsoft Azure Research – Systems
Redmond, WA, United States
inigog@microsoft.com

Hao Wang
Stevens Institute of Technology
Hoboken, NJ, United States
hwang9@stevens.edu

Rodrigo Fonseca
Microsoft Azure Research – Systems
Redmond, WA, United States
Fonseca.Rodrigo@microsoft.com

Abstract

Cloud platforms reduce deployment latency by pre-initializing resources, but fragmented resource management across cloud layers often leads to substantial over-provisioning and increased cost. Through production trace analysis and replay using state-of-the-practice resource management approaches, we show that independently managing resources at each layer results in a poor trade-off between resource waste and deployment latency.

We present Unipool, the first unified platform that exposes the state of pre-provisioned resources across layers, enabling more accurate resource pool sizing and dynamic multiplexing. Guided by insights from production traces, Unipool leverages cloud-native workload patterns to safely consolidate and oversubscribe pre-provisioned resources, significantly reducing platform cost while preserving quality of service. We implement Unipool on Kubernetes and Kata Containers, and evaluate it using both a large-scale event-driven simulator and a 25-VM testbed. Across mixed Function-as-a-Service and Container-as-a-Service production traces, we reduce resource waste by 92% while lowering container deployment latency by 21% compared with state-of-the-art baselines.

CCS Concepts: • Computer systems organization → Cloud computing.

Keywords: Cloud computing, container platforms, resource management

*Work done while interning at Microsoft Azure Research – Systems.



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, November 15-18, 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11.

<https://doi.org/10.1145/3828523.3857110>

ACM Reference Format:

Hanfei Yu, Enrique Saurez, Íñigo Goiri, Hao Wang, and Rodrigo Fonseca. 2026. Unipool: Efficient Management of Pre-Provisioned Resources for Container Platforms. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026)*, November 15-18, 2026, Hong Kong, China. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3828523.3857110>

1 Introduction

Modern services are designed to run natively in cloud environments, leveraging cloud resources, such as virtual machines (VMs), containers, and serverless functions, to efficiently scale with workload fluctuations. These services dynamically allocate resources, provisioning them only as needed to optimize costs [45]. For example, a web application may deploy additional container instances for a microservice during peak hours and deallocate them during off-peak periods. However, despite these advanced scaling capabilities, services often provision more resources than necessary due to inherent delays in creating and allocating cloud resources.

Provisioning cloud resources is not instantaneous; it incurs overhead from the cloud control plane as well as the time required to initialize and customize resources for individual clients. To reduce deployment latency, cloud platforms and clients commonly pre-initialize resources and reserve empty resource buffers, referred to as pre-provisioned resources (PPRs) in this paper. The specific strategies for pre-provisioning vary by resource type. For VMs, techniques such as snapshots [5, 21, 65] and warm pools [4, 48] are used. For serverless functions, keep-alive policies [24, 62] and container image caching [16, 72] are common. These techniques aim to remove expensive operations from the deployment critical path, improving responsiveness at the cost of increased idle resource consumption. For example, Azure Functions [51] reports that maintaining warm container pools accounts for over 76% of the platform cost [2].

Current limitations. While pre-provisioning significantly reduces deployment latency, it increases resource consumption and operational costs. Maintaining idle resources (e.g.,

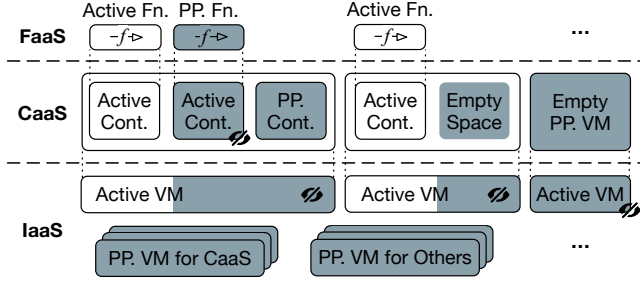


Figure 1. Layered dependency of pre-provisioned resources in a typical three-layer cloud platform, including Function-as-a-Service (FaaS), Container-as-a-Service (CaaS), and Infrastructure-as-a-Service (IaaS). “Fn.”, “Cont.”, and “PP.” denote function, container, and pre-provisioned, respectively. Gray regions indicate resource waste across the three cloud layers. Blind marks indicate that the corresponding resource waste is not visible to the current layer.

inactive containers or VMs in warm pools) introduces substantial cost overheads. Our analysis of production workloads from Azure shows that these policies can increase infrastructure costs by $2.6\times$ – $4.6\times$, with container warm pools in Function-as-a-Service (FaaS) and Container-as-a-Service (CaaS) architectures raising container-layer costs by up to $2.1\times$ and $6.9\times$, respectively. Some of these costs are necessary to hide deployment latencies, but others are artifacts of the layered architecture of modern cloud systems. Figure 1 shows how each “layer” (corresponding to cloud service abstractions such as FaaS, CaaS, and Infrastructure-as-a-Service (IaaS)) manages its own pre-provisioning independently, serving both internal and external clients. While this separation simplifies modular development, it leads to fragmented and inefficient resource management.

Cross-layer inefficiencies. The independent management of PPRs across cloud stack layers introduces two main inefficiencies: *excessive buffer sizing* and *limited resource multiplexing*. Lower layers, such as the VM platform, lack visibility into the number and state of PPRs maintained by higher layers. As a result, they conservatively assume all resources are fully utilized while many are idle and reserved for fast activation. This leads each layer to size its resource buffers for its own worst-case demand [2, 57], rather than for the true aggregate needs, and prevents lower layers from considering idle resources held by upper layers when sizing their own buffers. In addition, even when mechanisms like warm pools [2, 4, 48, 57] are available, they are typically siloed per customer, further amplifying over-provisioning and cost (see Section 2.4).

Moreover, even if PPR buffers were optimally sized, they are still treated with the same quality of service (QoS) as regular resources. Higher-level platforms lack access to low-level mechanisms for resource oversubscription, since they

do not directly allocate or manage server resources [2]. Meanwhile, lower layers cannot distinguish between active and idle PPRs, and thus provide uniform QoS to all resources. This lack of coordination prevents the stack from dynamically reallocating or oversubscribing resources based on actual usage, compounding inefficiency. In summary, the absence of cross-layer visibility and coordination leads to inflated buffer sizes and missed opportunities for efficient resource consolidation, highlighting the need for mechanisms that enable safe multiplexing and right-sizing of idle resources across the layers.

Our work. We propose Unipool, the first unified platform designed to reduce the cost of managing PPRs (*i.e.*, containers and VMs) while maintaining the same QoS for resource creation as current cloud platforms. Motivated by our detailed characterization of production traces from Azure, the core principle behind Unipool is to enhance visibility of PPRs across all layers in the cloud infrastructure and to leverage the unique properties of cloud-native workloads. To realize this shift, Unipool introduces new interfaces that enable differentiation between idle and active resources, allowing resource state information to be propagated to the appropriate resource managers. This cross-layer communication enables more accurate sizing and management of PPR buffers, ultimately reducing unnecessary over-provisioning and improving overall efficiency.

Unipool achieves cost-effective allocation of PPRs by exploiting the behavioral patterns of cloud-native applications. For instance, our analysis of container traces—focusing on container lifetimes and creation patterns—shows that containers typically have short lifespans and that their active periods rarely overlap across different customers. This observation suggests that containers can safely share the same underlying hardware, provided that resource creation latency is minimized, which can be accomplished through judicious oversubscription. To ground our analysis, we focus on the PPRs managed within the container infrastructure layer, considering both its upper layer (container consumers) and its lower layer (VM providers), to demonstrate the practical benefits of coordinated resource management.

We implement Unipool using open-source components, including Kata Containers [34] and Kubernetes [27]. Results on a 25-VM testbed and large-scale simulations driven by production traces demonstrate that Unipool reduces pre-provisioning cost by 92% while lowering deployment latency (*i.e.*, our key service level objective (SLO)) by 21% compared with state-of-the-art (SOTA) solutions.

Summary. We make the following main contributions:

- We characterize the SLO–cost trade-off of PPRs in container platforms with SOTA solutions.
- We propose Unipool, the first unified platform for managing PPRs that introduces new interfaces and mechanisms for efficient PPR oversubscription and orchestration.

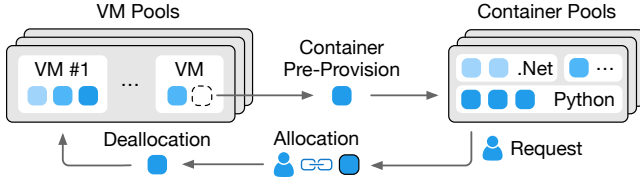


Figure 2. Container lifecycle in cloud platforms.

- We prototype Unipool on top of open-source frameworks, including Kata Containers and Kubernetes.
- We evaluate Unipool against SOTA baselines using mixed FaaS and CaaS production traces in large-scale simulations and on a 25-VM testbed.

2 Characterizing Pre-Provisioning

2.1 PPRs in Layered Clouds

Figure 6a shows a typical three-layer cloud architecture consisting of FaaS, CaaS, and IaaS. To satisfy deployment-latency SLOs [8–11, 50–53], each layer maintains *pre-provisioned resources* (PPRs), which trade additional resource usage for faster serving and execution [4, 22, 29, 48]. We refer to these resources as the PPRs of each cloud layer. Figure 2 illustrates the lifecycle of pre-provisioned containers in cloud platforms. Containers are created on VMs, pre-provisioned, and organized into image-specific container pools. Cloud platforms typically maintain multiple such pools, each corresponding to a container image. Upon request arrival, the platform first attempts to allocate a matching pre-provisioned container; otherwise, it falls back to a cold start.

Function-as-a-Service (FaaS). FaaS (sometimes referred to as serverless) platforms, such as AWS Lambda [9] and Azure Functions [51], maintain pools of warm containers to mitigate function cold starts [24, 59, 62, 72]. These containers typically include language runtimes but not user code or libraries [2, 6, 12]. The FaaS layer requests containers from the underlying CaaS platform.

Container-as-a-Service (CaaS). Container platforms, such as Azure Kubernetes Service (AKS) [52], Google Kubernetes Engine (GKE) [25], and AWS Elastic Kubernetes Services (EKS) [11], maintain both pre-provisioned containers and VM buffers to reduce deployment latency. When additional capacity is required, the CaaS layer requests VMs from the underlying IaaS provider.

Infrastructure-as-a-Service (IaaS). This layer provides VMs as the fundamental compute abstraction. To accelerate provisioning, cloud providers maintain pools of pre-provisioned VMs [8, 50]. These systems are generally agnostic to the workloads running inside the VMs and treat them as black boxes for resource management [14, 20, 29].

2.2 Layer Dependency of PPRs

Although each layer manages its PPRs independently, the resources are inherently dependent because cloud services are stacked hierarchically. The FaaS layer runs inside containers provided by CaaS, while the CaaS layer itself runs on VMs provisioned by IaaS. Figure 1 shows that this creates nested idle-resource dependencies across layers.

The IaaS layer considers a VM active whenever it hosts containers, even if those containers are merely idle warm pools maintained by CaaS. Similarly, the CaaS layer treats containers as active whenever they host functions, even if those functions are only warm instances maintained by FaaS. As a result, resources that are effectively idle at higher layers remain invisible to lower-layer schedulers.

This hidden idle waste reduces overall cloud efficiency. For example, a VM hosting only idle warm containers is still marked active by IaaS, preventing reclamation or consolidation. Likewise, containers hosting only warm serverless functions remain active from the perspective of CaaS. This paper identifies and eliminates such cross-layer inefficiencies for container platforms through coordinated PPR management across cloud layers.

2.3 Pre-Provisioned Resource Management

Existing cloud systems manage PPRs independently at each layer using several common policies:

Static pools. Some platforms maintain a fixed number of pre-provisioned instances to absorb bursts [2, 12]. For example, Azure Functions [2] provisions each container pool with a statically determined size to satisfy customer SLOs. While effective for latency, this approach wastes resources during low-utilization periods.

History-based autoscaling. Systems scale PPR pools based on historical workload and resource patterns [27, 62, 72]. For example, Kubernetes [27] Horizontal Pod Autoscaler (HPA) uses recent utilization windows for pod auto-scaling, while Azure Functions [62] predicts future invocations from recent request history.

Online pool size optimization. Some systems formulate pool sizing as an optimization problem and leverage Machine Learning (ML) models to make online provisioning decisions [57]. However, these approaches rely heavily on accurate workload modeling and can be sensitive to workload shifts and prediction errors.

Despite their differences, existing approaches optimize resource pools in isolation at individual cloud layers, ignoring cross-layer resource states. In this paper, we show that consolidating container pools across layers significantly improves platform-wide resource efficiency while maintaining low deployment latency.

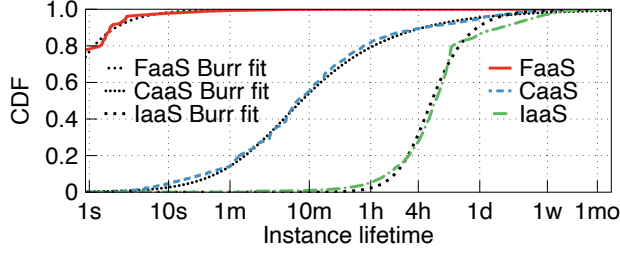


Figure 3. Instance lifetimes across the FaaS (Azure Functions traces [62]), CaaS, and IaaS layers. FaaS and CaaS report container lifetimes, while IaaS reports lifetimes of the hosting VMs.

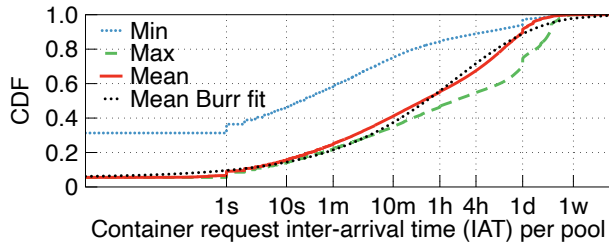


Figure 4. Per-pool container request inter-arrival times (IATs) in our CaaS production traces. We report the CDF of different percentiles computed from each pool’s IAT distribution.

2.4 Production Trace Analysis

To quantify PPR overheads, we analyze production traces collected from Azure container platforms. The dataset spans containers and their underlying VMs, capturing cross-layer dependencies between demand and provisioning. We augment these with Azure Functions traces [62] to study idle-resource overheads under common PPR policies and their latency–resource trade-offs.

Production trace collection. Our dataset includes container lifetimes and their hosting VMs across Azure’s global infrastructure. It covers multiple regions over one week (April 1st–7th, 2026). To reduce collection cost, we restrict traces to container lifetimes of at most one week, capturing over 95% of containers. We release a sanitized subset of our production traces from Azure container platforms at <https://github.com/Azure/AzurePublicDataset>.

Temporal behavior. We model container and VM dynamics by fitting candidate distributions (Burr, Weibull, Lognormal, Gamma, and Exponential) to empirical cumulative distribution functions (CDFs) and selecting the best fit via the Kolmogorov–Smirnov statistic [46].

Instance lifetimes. Figure 3 shows lifetime distributions. All traces are best fit by Burr distributions. Containers are short-lived (FaaS: 50% <3 seconds and 90% <1 minute; CaaS: 55%

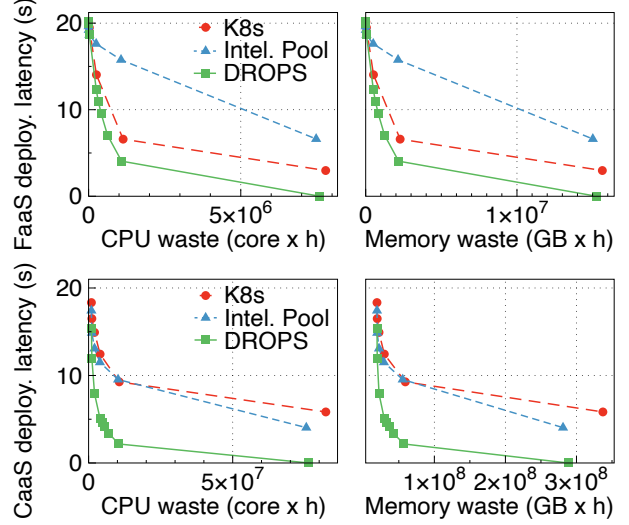


Figure 5. Deployment latency vs. PPR waste trade-off under FaaS and CaaS production traces. **K8s**: threshold-based pool sizing [27]. **Intel. Pool**: ML-based resource pool optimization used in Azure Synapse [57]. **DROPS**: static resource pool configuration from historical workload traces [2].

<10 minutes and 82% <1 hour), while VMs persist longer (80% >3 hours). This lifetime gap directly creates idle periods where VM resources remain unused.

Request arrivals. Figure 4 shows inter-arrival times (IATs) of container pools grouped by container image, also best fit by a Burr distribution. Requests are infrequent (75% >1 minute, 45% >1 hour), further amplifying idle periods between container executions.

Latency–resource trade-off. Using these traces, we evaluate three SOTA PPR management solutions with a high-fidelity, three-layer model (FaaS, CaaS, IaaS). Details are in Section 5.1. We evaluate three performance metrics that directly reflect resource efficiency and customer SLOs: (1) *Deployment latency*, the average container deployment time; (2) *CPU waste*, the idle CPU-time aggregated across VMs; and (3) *Memory waste*, the idle memory-time aggregated across VMs. We use the product of unallocated resources and time following prior work [62, 63, 72].

Figure 5 shows the trade-offs between average container deployment performance and resource waste time across three baseline solutions (Section 2.3). We sweep their key system configuration parameters to generate variants: the autoscaling threshold of container pools (*K8s* [27]), tuning knob of pool optimization (*Intel. Pool* [57]), and pool size (*DROPS* [2]). Existing PPR solutions exhibit a clear trade-off: improving deployment performance requires overprovisioning, while reducing cost degrades performance, preventing an optimal balance.

Insights. The upper layer (containers) incurs substantial resource waste on the lower layer (VMs) due to short container lifetimes and infrequent request arrivals, amplifying the resource inefficiency in the cloud. Managing PPRs at a single layer cannot achieve an optimal balance between deployment latency and resource cost. Effective optimization of resource efficiency requires coordination and consolidation across layers.

3 Unipool

3.1 Objectives and Challenges

Improve resource efficiency across layers. Due to the direct infrastructural cost of resource provisioning, improving resource efficiency has long been a central goal for cloud platforms [14, 18, 20, 29, 62, 76, 77]. However, increasing resource density across the cloud stack remains challenging due to fragmented resource pools across layers. Unipool reduces PPR waste by accurately sizing and oversubscribing container pools across different cloud layers.

Minimize deployment latency. Containers [1, 28, 38, 47] have become the de facto building block for serving CaaS and FaaS workloads in the cloud. For example, AWS Fargate and Lambda employ Firecracker [1]. While reducing PPR waste improves infrastructural cost efficiency, aggressive resource consolidation can significantly increase deployment latency [20, 57]. Moreover, heterogeneous workloads (e.g., from FaaS and CaaS) exhibit diverse characteristics, including request arrival patterns, request lifetimes, and resource demands, which must be carefully managed by the cloud provider. We design Unipool to minimize container deployment latency for heterogeneous cloud workloads while reducing PPR waste and maintaining customer SLOs.

3.2 Workflow of Unipool

We design Unipool to efficiently manage and consolidate PPRs across traditional multi-layer cloud systems. Figure 6b shows the architecture and workflow of Unipool, which consists of three steps to optimize system-wide PPR utilization:

Step ①: Resource pool consolidation. Unipool first merges container and VM pools and right-sizes them based on *upper-layer resource signals*. In contrast to existing PPR management solutions that operate independently at each layer, Unipool receives resource signals (e.g., the ratio of active allocations to total provisioned resources) from upper layers to guide the container and VM pool sizing decisions at each layer (Section 3.3).

Step ②: Container oversubscription. After properly sizing PPR pools at each layer, Unipool further improves resource utilization through adaptive PPR oversubscription. We leverage upper-layer resource signals to oversubscribe without increasing container deployment latency. For example, the CaaS layer provides containers to the FaaS layer

assuming that all containers will run FaaS workloads. However, FaaS may use some containers to run workloads while leaving others idle to mitigate cold-starts. Unipool records this resource status in the resource signals and adaptively oversubscribes these hidden idle resources, thereby consolidating overall system resource allocations (Section 3.4). Note that we only oversubscribe *pre-provisioned* resources (i.e., pre-provisioning phase in Figure 2), in contrast to existing works that overbook or overcommit active or running resources [7, 27, 35, 67]. As a result, applications are always allocated their full requested resources at runtime, avoiding interference with execution performance.

Step ③: Container orchestration. When a request arrives, a pre-provisioned container must be activated and restored to its full resource allocation, which may require reclaiming resources from other oversubscribed containers through eviction. To minimize eviction overhead, Unipool exploits the temporal multiplexing opportunities of PPRs, as pre-provisioned containers are activated infrequently and are typically short-lived (Section 2.4). Hence, Unipool adaptively places and co-locates containers with different sizes and oversubscription levels across VMs to further consolidate resource allocations. However, excessive consolidation can trigger substantial container evictions and reallocations when heavily oversubscribed containers in hot pools are activated [7, 35]. This introduces a fundamental trade-off: consolidating too many oversubscribed containers improves resource utilization but risks higher deployment latency due to eviction and reallocation, while loose oversubscription can result in high resource waste. To address this challenge, Unipool introduces novel container placement (③a), activation (③b), and eviction strategies (③c) that minimize resource eviction and reallocation, thereby reducing infrastructural cost, operational overhead, and deployment latency (Sections 3.5 to 3.7).

3.3 Resource Pool Consolidation

Container pool consolidation. In Unipool, container pool management is unified at the CaaS layer, regardless of the upper-layer service type (e.g., AWS Lambda [9] or AKS [52]). Existing FaaS container pool management approaches [2, 24, 62, 72] maintain pre-provisioned container pools to mitigate function cold starts by requesting containers from the underlying CaaS platform. Meanwhile, CaaS platforms also maintain their own container pools, commonly as pods [11, 27, 52], to serve CaaS workloads. Unipool simplifies container pool management by pushing FaaS container pools down to the CaaS layer. Figure 6b shows that Unipool does not maintain separate container pools at the FaaS layer. Instead, it directly activates and allocates containers from the unified CaaS-layer pool when serving function invocations.

VM pool consolidation. Unipool further consolidates VM pools across the CaaS and IaaS layers. While containers in

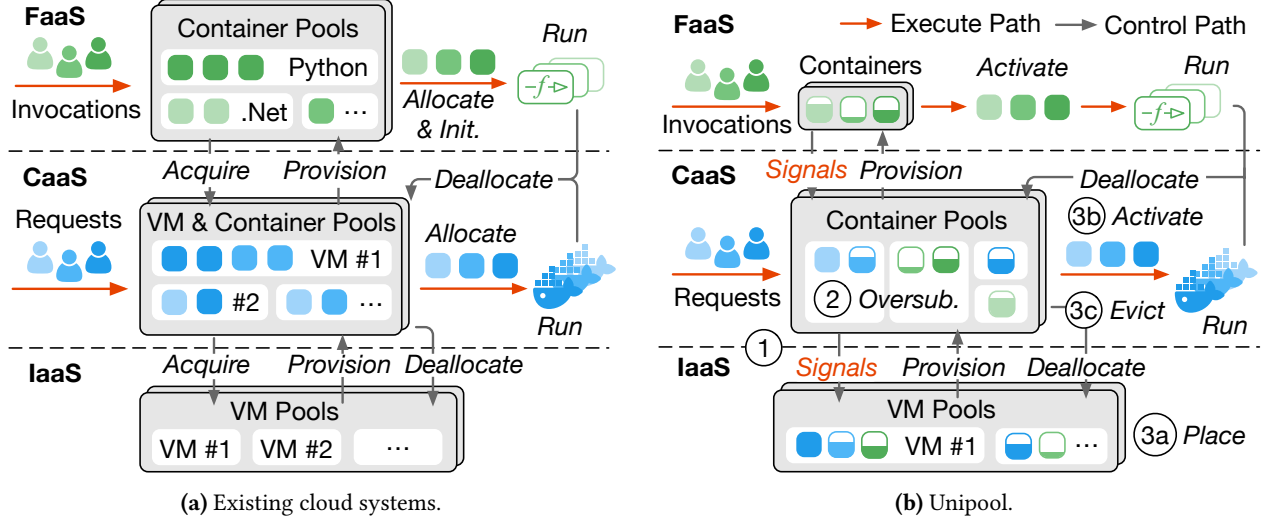


Figure 6. Architecture and workflow of existing multi-layer cloud systems and Unipool. The pre-provisioned resource (PPR) pools (*i.e.*, containers and VMs) are consolidated and unified across layers for efficient management in Unipool. Partially filled containers indicate oversubscription of pre-provisioned containers.

CaaS run atop VMs in IaaS, existing systems often maintain separate VM pools for each [42, 57]. In contrast, Unipool unifies them into a single pool, analogous to its unified container pool design (Figure 6).

VM autoscaling. Existing VM autoscaling methods are typically reactive, relying on coarse-grained utilization or allocation thresholds (*e.g.*, Kubernetes scale-up/down [27]). In contrast, Unipool exposes fine-grained *resource signals* from upper layers to improve pool sizing. Rather than relying solely on aggregated container allocations, Unipool accounts for the intended footprints of container pools in both FaaS and CaaS layers.

At each autoscaling interval t , Unipool computes CPU and memory demand from active and preprov containers:

$$\mathbf{R}_{\text{vm}}(t) = \mathbf{R}_{\text{active}}(t) + \alpha \cdot \mathbf{R}_{\text{preprov}}(t), \quad \mathbf{R} = (\text{CPU}, \text{Mem}),$$

where $\mathbf{R}_{\text{active}}(t)$ is the aggregated allocation of actively running containers, $\mathbf{R}_{\text{preprov}}(t)$ is the aggregated allocation of pre-provisioned containers, and $\alpha \in [0, 1]$ controls how strongly pre-provisioned capacity contributes to VM pool sizing. When $\alpha = 0$, Unipool sizes VM pools aggressively by ignoring PPR demands and accounting only for active containers. When $\alpha = 1$, Unipool conservatively reserves sufficient VM capacity for both active and pre-provisioned containers, which is equivalent to existing VM autoscaling.

3.4 Container Oversubscription

To further improve container resource efficiency, we design Unipool to dynamically oversubscribe pre-provisioned container pools at the CaaS layer based on per-pool request arrival and activation patterns. Container pools are typically organized and indexed by customer metadata [2, 57, 62], such

as customer IDs, container images, and resource limits. Containers within the same pool are homogeneous with respect to these metadata. We present the design of container pool oversubscription below.

Request arrival pressure (AP). For each container pool p , we define a dimensionless metric, AP:

$$AP(p) := \frac{T_p}{IAT(p)}, \quad p \in [1, \dots, P],$$

where T_p is the container startup latency, and $IAT(p)$ is the inter-arrival time of requests to pool p . Intuitively, AP captures the expected number of requests arriving during a cold-start interval when no container is available, thus reflecting the potential deployment latency impact. Cold pools tend to have low AP, while hot pools exhibit high AP. We periodically recompute AP at every interval t (*e.g.*, 30 s) and apply exponentially weighted moving average (EWMA) smoothing to stabilize IAT estimation.

Container minimum resource reservation (MRR). We define a MRR for each pool to guarantee a baseline allocation for essential functionality (*e.g.*, OS requirements):

$$MRR(p) := \max \left(\frac{cpu_{\min}}{cpu_p}, \frac{mem_{\min}}{mem_p} \right), \quad p \in [1, \dots, P],$$

where $cpu_{\min}$ and $mem_{\min}$ are system-level minima, and cpu_p and mem_p are per-container limits in pool p . Larger containers yield lower MRR and thus greater oversubscription potential, while smaller ones have higher MRR and are less amenable to oversubscription. The minimum requirements are system constants independent of container size. Hence, the maximum oversubscription factor of container pools is effectively capped by MRR as $\frac{1}{MRR}$.

Per-pool oversubscription factor. We compute the oversubscription factor $F(p)$ for each pool based on AP and MRR:

$$F(p) := \frac{1}{MRR(p) + (1 - MRR(p)) \cdot \left(1 - \exp\left(-\frac{AP(p)}{k}\right)\right)},$$

where $\exp$ denotes the exponential mapping function that bounds the term within $(0, 1]$, and k is a *scaling factor* controlling the aggressiveness of oversubscription. Intuitively, $F(p)$ is jointly determined by MRR and AP. Higher MRR and AP lead to more conservative oversubscription to avoid frequent container evictions and cold starts, while lower values enable more aggressive oversubscription with limited impact on deployment latency.

Scarcity brake. To prevent rapid depletion of a container pool, we introduce a scarcity brake mechanism. When the available capacity of a pool falls below a predefined threshold, we immediately disable oversubscription for that pool and set $F(p) = 1$ for future container allocations. Specifically, for each container pool $p \in [1, \dots, P]$, we define an AP-aware brake threshold B_p :

$$B(p) := B_{\max} \cdot \left(1 - \exp\left(-\frac{AP(p)}{k}\right)\right), \quad p \in [1, \dots, P],$$

where $B_{\max}$ is the maximum system-wide brake threshold and k is the scaling factor. Pools with higher AP deplete faster, thus requiring higher thresholds to enforce more conservative oversubscription. This mechanism applies only to newly created containers, while existing oversubscribed containers remain unaffected to avoid operational overhead.

3.5 Container Placement

Commercial cloud platforms typically use “best-fit” placement [13, 29, 69, 76], the de facto scheduling approach for minimizing server/VM count and maximizing utilization [32]. Building on this, Unipool adopts a simple yet effective policy that consistently outperforms vanilla best-fit across oversubscription settings. Specifically, Unipool considers the following three factors when placing a container among multiple VM candidates:

Placement feasibility. Unipool first performs an admission check on candidate VMs. A feasible VM must provide sufficient capacity for the incoming container and guarantee that oversubscribed containers can reclaim their reserved resources upon activation. Unipool therefore filters the set of physically eligible VMs for container placement.

VM load. Among feasible candidates, Unipool prefers the most heavily loaded VMs, up to a predefined allocation threshold to avoid severe resource contention [29], thereby achieving compact best-fit packing and minimizing server provisioning cost.

Container pool diversity. For VMs with similar load levels, Unipool favors those VM candidates hosting more diverse

container pools. This reduces the likelihood that bursty workloads from a single customer disproportionately impact a VM and trigger large-scale container evictions.

3.6 Container Activation

When multiple pre-provisioned containers are available in the pool, we design Unipool’s container selection and activation policy to be *eviction-aware*. Activating an oversubscribed container reclaims its full resource allocation and may trigger the eviction of other pre-provisioned containers to satisfy VM capacity constraints. However, container eviction is a major source of deployment cold starts. To select among multiple eligible pre-provisioned container candidates, we first identify their host VMs and estimate the number of container evictions required for each activation. Unipool then prioritizes activating the candidate that incurs the fewest evictions. If multiple candidates result in the same eviction count, we break ties by selecting the host VM that retains the largest amount of remaining free capacity after activation and eviction.

3.7 Container Eviction and Pool Hedging

Urgency-aware eviction. When a VM becomes overcommitted after activating a container due to physical resource reclamation, Unipool must evict one or more pre-provisioned containers so that the VM capacity is not exceeded. Instead of classic cache eviction policies that primarily consider historical reuse (e.g., least recently used (LRU) or least frequently used (LFU)), Unipool ranks eviction container candidates by computing AP-aware urgency scores. For each pool p , Unipool computes an eviction score:

$$E(p) := \frac{S(p)}{\max(AP(p), \epsilon)}, \quad p \in [1, \dots, P],$$

where $S(p)$ is the fraction of ready pre-provisioned containers above pool p ’s target size, $AP(p)$ is the arrival pressure defined in Section 3.4, and ϵ is a trivial constant that avoids division by zero. Intuitively, higher $E(p)$ means the pool has more spare capacity relative to its near-term demand, and is therefore safer to evict from. This protects high-AP pools even when they temporarily contain idle containers, while allowing Unipool to reclaim capacity from low-AP pools with surplus pre-provisioned containers. When ranking candidates on the same VM, Unipool normalizes $E(p)$ across the candidate set and combines it with normalized container size to prefer evictions that both avoid high-AP pools and quickly reclaim the overloaded resources.

Container pool hedging. Unipool uses a hedging strategy that selectively maintains over-provisioned container pools to hedge against bursty request arrivals and deployment cold starts. For each container pool $p \in [1, \dots, P]$, we introduce a hedge buffer that prevents individual container evictions from immediately reducing the pool below its target size. Similar to the oversubscription mechanism in Section 3.4,

we design the hedge size H_p for each container pool to be AP-aware:

$$H(p) := H_{\max} \cdot \left(1 - \exp\left(-\frac{AP(p)}{k}\right)\right), \quad p \in [1, \dots, P],$$

where $H_{\max}$ denotes the maximum system-wide hedge size and k is the scaling factor. Intuitively, pools with higher AP are more vulnerable to burst-induced depletion and therefore maintain larger hedge buffers to reduce unexpected deployment cold starts.

4 Implementation

We prototype Unipool on Kata Containers [34] and Kubernetes [27], the container orchestration stack underlying commercial platforms such as AWS EKS [11], GKE [25], and AKS [52]. The prototype runs on Azure Linux 3 VMs with modifications to three components:

- **PPR controller** that monitors PreProvisionPool custom resources and transitions pods through three activation states (pending $\rightarrow$ booting $\rightarrow$ active) via annotations, implementing pool sizing, the activation policy of Section 3.6, and VM autoscaling.
- **Kubelet patch** that updates Kata cgroup limits [40] at each transition. It assigns pods a boot envelope of 250 millicores and 256 MiB for fast startup, an idle envelope of 100 millicores and 64 MiB to satisfy per-pool MRR (Section 3.4), and the full requested resources only upon activation.
- **Scheduler plugin** that bin-packs against the oversubscribed view in Section 3.4 rather than raw pod requests, decoupling resource accounting from cgroup enforcement to keep oversubscription independent of execution.

The container network interface, runtime, image format, and the rest of the control plane remain unmodified.

5 Evaluation

5.1 Experimental Setup

Real cluster. We use our Unipool implementation on a Kubernetes cluster with 25 Azure D16s_v3 VMs. Each VM has 16 CPUs and 64 GiB memory with Linux 6.6.138, containerd 2.1.6, Kata 3.19.1 over Cloud Hypervisor, and Cilium 1.16.5. The cluster has one control-plane node, one dedicated evaluation-driver node excluded from scheduling, and 23 worker nodes, totaling 368 worker cores. The evaluation driver runs in the same virtual network but outside the worker pool, ensuring that neither trace replay nor in-pod workloads interfere with measurements. The testbed further includes a VM autoscaler operating at 15 s intervals with a minimum pool size of three VMs. Container pools are sized using DROPS [2] with a P100 target, the most demanding configuration in which every request must be served from a PPR, thereby maximizing oversubscription pressure.

Table 1. Cluster-wide resource and cost comparison.

Metric	No oversub.	Unipool	Reduction
PPR CPU core hours	672.35	121.22	−82.0%
Worker VM hours	64.46	16.00	−75.2%
3-hour VM cost (\$)	\$49.50	\$12.29	−75.2%

Simulator. Following prior studies on large-scale cloud production trace evaluation [2, 57, 62, 76], we build a high-fidelity event-driven simulator to evaluate Unipool and SOTA baselines. The simulator is highly extensible and supports arbitrary numbers of services across the FaaS, CaaS, and IaaS layers. Container requests from production traces are encapsulated as events with metadata, including request IDs, image IDs, execution durations, and resource limits. These events are injected into the simulator to faithfully replay realistic container allocation and management behavior. We use an average container cold-start delay of 20 s, consistent with reported startup measurements on commercial container platforms [41, 68].

Baselines. We primarily compare Unipool against four SOTA baselines: (1) *K8s* [27], specifically the HPA in Kubernetes that sizes the container pools based on predefined thresholds; (2) *Intelligent Pooling* [57], an ML-based approach that dynamically optimizes pool sizes using workload prediction; and (3) *DROPS* [2], which statistically determines pool sizes based on historical traces given target SLOs. (4) *No pre-provision*, which disables all pre-provisioned containers and creates containers on demand, serving as a lower bound on resource waste measurement. All baselines are evaluated in the simulator, whereas the real-cluster evaluation primarily compares Unipool with a no-oversubscription configuration (Section 5.2). The cluster experiment focuses on container-resource oversubscription because its runtime effects cannot be evaluated confidently through simulation alone. On our 25-VM testbed, we replay a three-hour subset of the production trace with customer-level downsampling to evaluate Unipool’s oversubscription in a real cluster.

Workload traces. Unlike prior work [2, 44, 57, 62, 63, 72], which evaluates a single trace type (*i.e.*, either FaaS or CaaS), Unipool is the first to study mixed FaaS and CaaS production traces for multi-layer resource management.

For real-cluster experiments, we replay a three-hour trace segment containing 947 invocations from the second-day peak of the traces from Azure container platforms. The workload is sampled at the customer level such that all pods fit on a single worker while exceeding the boot envelope. The trace events are replayed as POST /acquire and DELETE requests. Each invocation executes `stress-ng --cpu N` for its trace-derived lifetime, where N equals the pod CPU request rounded up to the nearest core. For large-scale simulation,

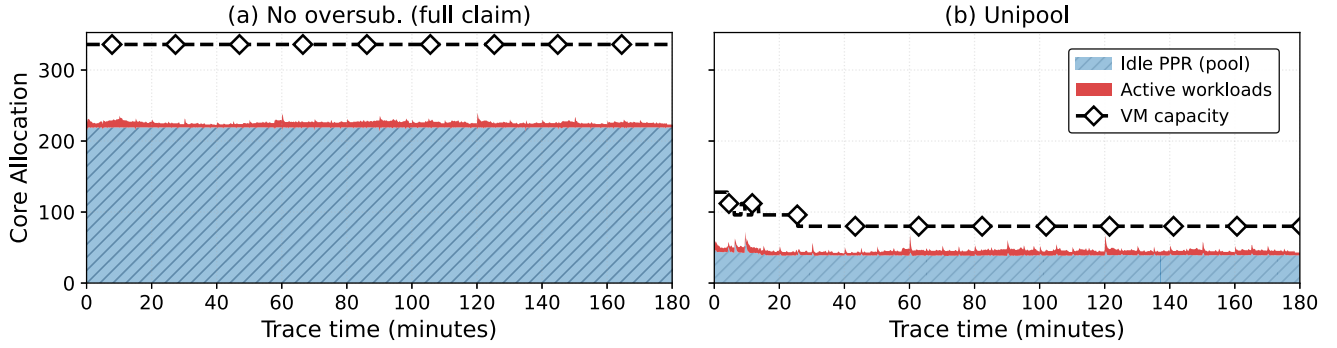


Figure 7. Cluster-wide CPU utilization timeline. Core allocation during the three-hour trace replay on the 25-VM cluster using No oversubscription (left) vs. Unipool (right).

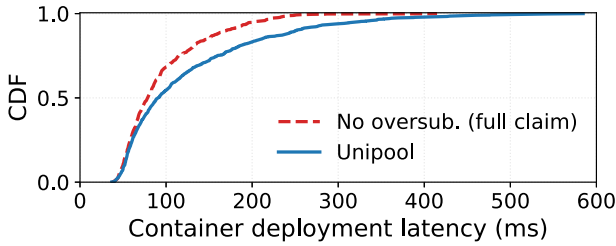


Figure 8. Container deployment latency CDF over the three-hour trace replay on the 25-VM cluster.

we use the same FaaS and CaaS traces as in Section 2.4, align their timelines, and merge them as input to the simulator.

Metrics. We focus on two key metrics. (1) *Container deployment latency*: the time elapsed from request arrival until the container is allocated and ready to serve the request. This metric reflects customer-facing SLOs and service responsiveness. (2) *Resource waste time*: the product of allocated resources and their occupation duration, which quantifies infrastructure resource cost for cloud providers [62, 63, 72]. We primarily measure CPU and memory waste and leave other resource types (e.g., storage and network bandwidth) to future work. We also report fine-grained system behavior, including scheduling decisions and runtime overheads.

5.2 Real-Cluster Evaluation

We compare two configurations on the same cleaned cluster: (1) a *no-oversubscription* baseline, where PPRs are assigned their full resource requests, and (2) Unipool with a scaling factor of $k = 10$, corresponding to an aggressive oversubscription setting in the real-cluster deployment and chosen to stress-test Unipool under high consolidation pressure.

End-to-end results. Figure 7 plots the cluster-wide CPU timeline. The no-oversubscription baseline maintains a near-constant PPR footprint of ~ 224 cores based on the DROPS-P100 estimate. In contrast, Unipool reduces the idle PPR

footprint to ~ 40 cores and enables the autoscaler to reclaim VMs during demand valleys, reducing the number of active workers from 23 to 7. The cluster-wide memory allocation timeline follows the same trend as the CPU timeline.

Figure 8 reports container deployment latency CDFs and Table 1 summarizes cluster-wide infrastructure cost. Across 947 invocations, Unipool achieves a 100% PPR hit rate, matching the no-oversubscription baseline. Despite reserving $5.5\times$ less PPR capacity, Unipool remains within 212 ms at P99 deployment latency (461 ms vs. 249 ms; P50 91 ms vs. 79 ms). At the same time, Unipool reduces PPR CPU hours by 82% and worker VM hours by 75%, lowering the three-hour infrastructure cost from \$49.50 to \$12.29.

Cold-start comparison. A cached-image cold start in this setup takes 4.4 s at the median and 7.2 s at P99, compared with 90 ms and 459 ms under Unipool, corresponding to a $49\times$ median speedup and a $16\times$ P99 speedup. Figure 8 shows the latency CDF and further confirms that oversubscription introduces a modest tail-latency overhead relative to the no-oversubscription baseline. Meanwhile, the cluster CPU timeline in Figure 7 shows substantially smaller PPR footprints and earlier VM reclamation under Unipool, yielding large resource savings at minimal activation cost.

5.3 Large-Scale Simulation

Container resource waste and deployment latency. We replay our large-scale production trace in our simulator to compare Unipool against the SOTA container management solutions. Figure 9a breaks down the wasted resources and average container deployment latency. Across both FaaS and CaaS traces, Unipool substantially reduces resource waste compared to all baselines. Excluding IaaS empty waste observed without pre-provisioning (No PP.), Unipool reduces CPU and memory waste by 92% over DROPS-P98. The reduction comes from two sources. First, Unipool decreases waste from pre-provisioned containers (FaaS PP. and CaaS

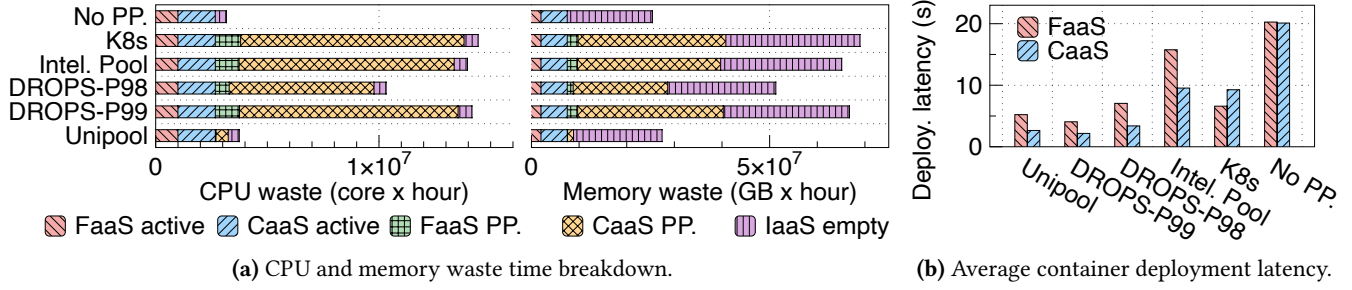


Figure 9. Resource waste decomposition and deployment latency comparison. (a) CPU and memory waste time broken down into active container executions (*FaaS* and *CaaS* active), pre-provisioned container resources (*FaaS* and *CaaS* PP), and fragmented unused VM capacity (*IaaS* empty). Baselines include Kubernetes HPA [27] (*K8s*), Intelligent Pooling [57] (*Intel. Pool*), DROPS [2], and no pre-provisioning (*No PP*). (b) Average container deployment latency for *FaaS* and *CaaS* traces.

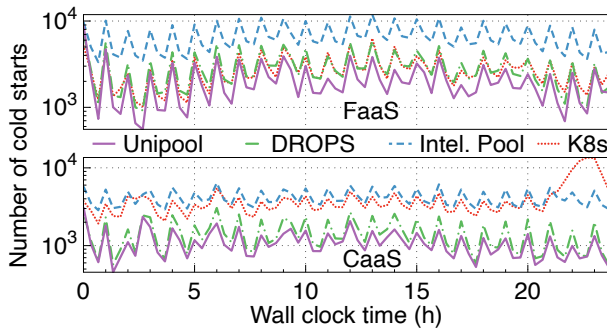


Figure 10. Container deployment cold-start timeline during a 24-hour replay of mixed *FaaS* and *CaaS* production traces.

PP.) through container pool consolidation and dynamic over-subscription across cloud layers. Second, Unipool reduces fragmented unused VM capacity (*IaaS* empty) through efficient container packing, activation, and eviction policies. The trends mirror the real-cluster results in Section 5.2 at large scale.

The resource waste of Unipool is close to the *No PP*. baseline, which represents the lower bound without maintaining PPRs. While Unipool and DROPS-P99 use the same pool-sizing target, Unipool reduces CPU and memory waste by 95% while achieving comparable deployment latency. Lowering the DROPS target SLO from P99 to P98 decreases resource waste, but at the cost of deployment latency that is higher than Unipool across both *FaaS* and *CaaS* workloads. These results indicate that Unipool achieves better trade-offs with efficient cross-layer PPR management and oversubscription.

Despite using significantly fewer PPRs, Unipool achieves the lowest average deployment latency across both *FaaS* and *CaaS* workloads. Compared with DROPS-P98, Intelligent Pooling, and *K8s*, Unipool reduces average deployment latency by 26.0%, 66.9%, and 20.9% for *FaaS* workloads and 21.9%, 72.2%, and 71.4% for *CaaS* workloads. In contrast, while *No PP*. eliminates nearly all pre-provisioning waste, it

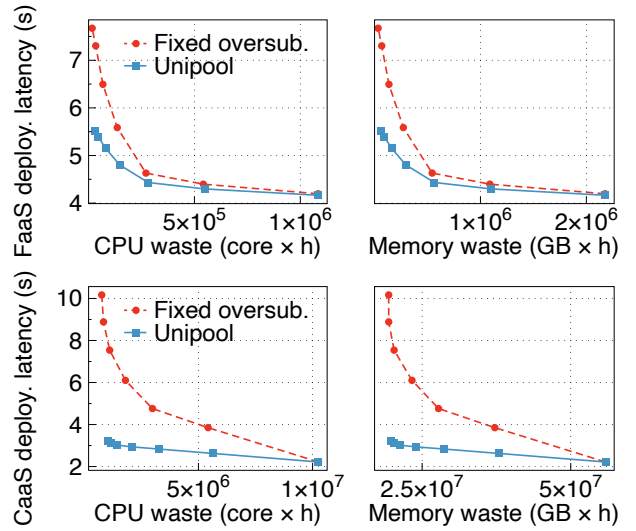


Figure 11. Container oversubscription comparison. Average container deployment latency vs. resource waste using *FaaS* and *CaaS* traces, with oversubscription factor decreases from 64 to 1 (left to right).

suffers from $3.9\times$ and $7.6\times$ higher deployment latency than Unipool for *FaaS* and *CaaS* workloads, respectively.

Container deployment timeline. Figure 10 shows the number of container deployment cold starts over a 24-hour replay period in the second-day traces. Across both *FaaS* and *CaaS* traces, Unipool consistently maintains the lowest cold-start counts among all evaluated solutions. On average, Unipool reduces deployment cold starts by 24.4%, 68.7%, and 28.8% compared with *K8s*, Intelligent Pooling, and DROPS-P98 for *FaaS* workloads, respectively. For *CaaS* workloads, Unipool achieves even larger reductions of 72.4%, 73.3%, and 23.7%. Specifically, during workload bursts, Unipool rapidly adapts to fluctuations in both *FaaS* and *CaaS* request arrivals, preventing the cold-start spikes observed in existing approaches.

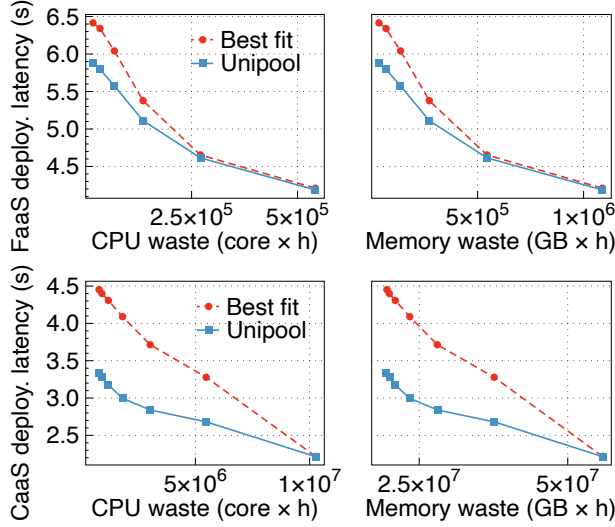


Figure 12. Container placement comparison. Average container deployment latency vs. resource waste using FaaS and CaaS traces under different placement strategies.

5.4 Ablation Study

In Figures 11 to 13 and 15, each comparison replaces one Unipool mechanism while retaining the others, so differences from full Unipool isolate that mechanism’s contribution. Figure 11 replaces adaptive oversubscription with fixed oversubscription; Figure 12 replaces Unipool’s container placement with vanilla best-fit placement; Figure 13 replaces its activation policy with random or round-robin activation; and Figure 15 replaces its eviction policy with largest-first (LF), greedy dual (GD), least-recently-used (LRU), or least-frequently-used (LFU) eviction. Figure 14 shows the correlation between eviction volume and deployment latency under fixed oversubscription factors.

Container oversubscription. Figure 11 shows the trade-off between deployment latency and resource waste for fixed oversubscription, which is the common practice in clouds [15, 37], and Unipool on both real-world FaaS and CaaS traces. For the fixed strategy, we vary the oversubscription factor from 64 to 1 with an exponential step of 2 (e.g., 64, 32, . . . , 1). For Unipool, we vary the max factor m over the same range. Unipool significantly reduces resource waste while achieving similar or lower container deployment latency for both FaaS and CaaS workloads. Specifically, Unipool reduces CPU and memory waste by 4.6× and 4.5× for FaaS workloads, and reduces CPU and memory waste by 5.2× and 1.9× for CaaS workloads, while maintaining lower deployment latency.

Container placement. Figure 12 shows the trade-off between deployment latency and resource waste at the FaaS and CaaS layers using their corresponding production traces. The results show that, compared to the vanilla best-fit placement strategy, Unipool achieves similar or lower resource

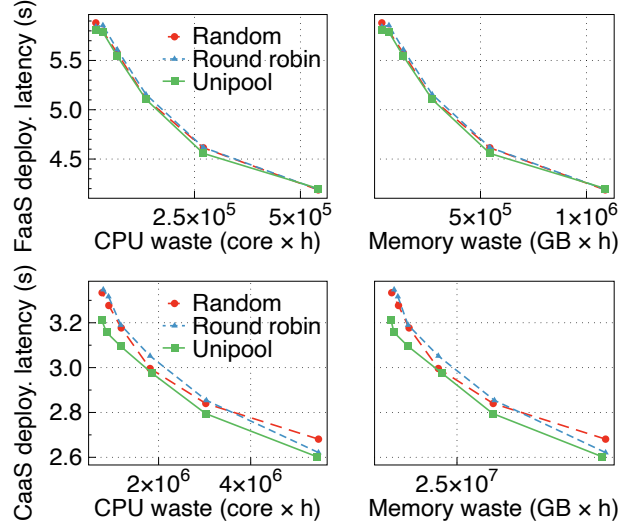


Figure 13. Container activation comparison. Average container deployment latency vs. resource waste using FaaS and CaaS traces across activation strategies.

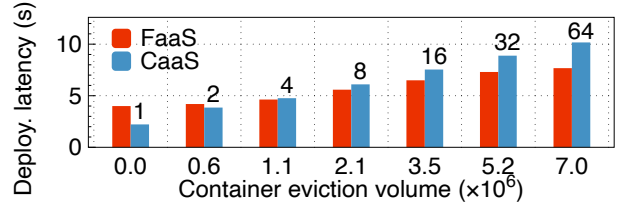


Figure 14. Correlation between container deployment latency and eviction volume under fixed oversubscription factors ranging from 1 to 64 when running FaaS and CaaS traces.

waste while consistently delivering lower container deployment latency across both FaaS and CaaS workloads.

Container activation. Figure 13 compares the trade-off performance of Unipool against two alternative activation strategies under Unipool’s oversubscription mechanism. We vary the maximum oversubscription factors from 64 to 2 to evaluate activation behavior across different contention levels: (1) *Random*, which randomly selects a pre-provisioned container for activation. (2) *Round robin*, which collects the host VMs of available candidates and activates containers in a cyclic manner across VMs. Across most oversubscription settings, Unipool consistently achieves a better deployment latency versus resource waste trade-off, providing lower deployment latency while maintaining similar or lower resource waste.

Container eviction. Figure 14 shows the relationship between container eviction volume and deployment latency under different oversubscription factors using both FaaS and CaaS traces. The results reveal a strong correlation between

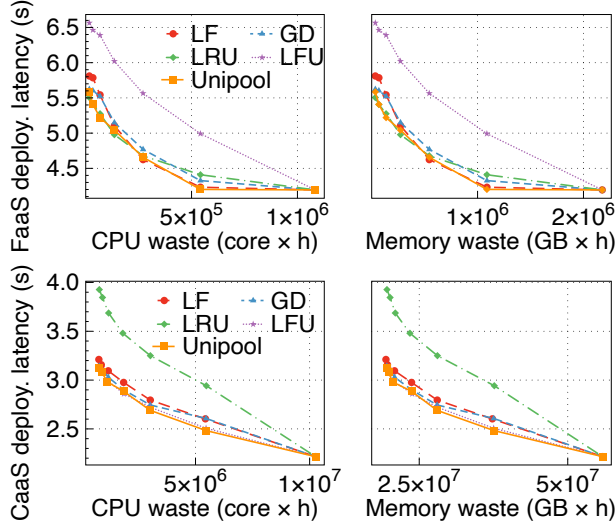


Figure 15. Container eviction comparison. Average container deployment latency vs. resource waste using FaaS and CaaS traces across eviction strategies.

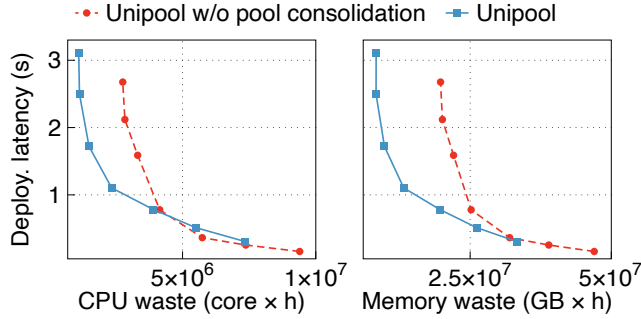


Figure 16. Pool consolidation comparison. Average container deployment latency vs. resource waste for Unipool and its variant with pool consolidation disabled.

eviction volume and deployment latency, where higher eviction volume consistently leads to longer deployment latency.

Figure 15 compares Unipool against four representative eviction policies commonly used in caching systems: (1) *Largest first (LF)*, which evicts containers that reclaim the largest amount of resources. (2) *Greedy dual (GD)*, which prioritizes containers based on a combination of startup cost, resource size, and activation frequency, following prior cloud cache systems [24, 63, 71]. (3) *Least recent used (LRU)*, which evicts the least recently used containers. (4) *Least frequently used (LFU)*, which evicts the least frequently used containers.

Across both FaaS and CaaS traces, Unipool consistently achieves the best or near-best deployment latency–resource waste trade-off. While LF and GD focus on immediate resource reclamation efficiency and LRU/LFU rely solely on historical access patterns, Unipool jointly considers resource

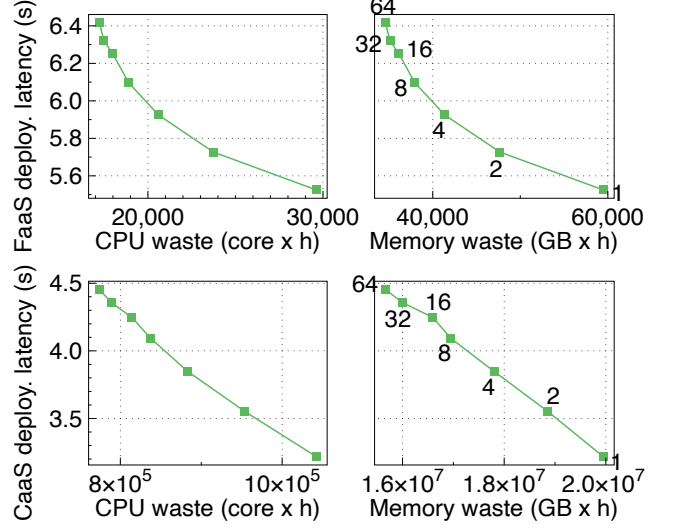


Figure 17. Sensitivity analysis. Average container deployment latency vs. resource waste using FaaS and CaaS traces under different scaling factors k .

recovery and arrival pressure. Consequently, Unipool avoids evicting containers that tend to be activated in the near future, reducing eviction-induced cold starts and enabling lower deployment latency at comparable resource wastes.

Pool consolidation. Figure 16 compares Unipool and its variant with pool consolidation disabled across oversubscription factors. Unipool substantially reduces resource waste: at factor 64, CPU and memory waste decrease by 60.0% and 57.7%, respectively, with a modest increase in container deployment latency.

5.5 Sensitivity Analysis

The primary performance knob of Unipool is the scaling factor k (Section 3.4), which controls the aggressiveness of resource oversubscription, activation, eviction, and hedging, thus balancing between deployment latency and resource waste. Figure 17 evaluates Unipool under different values of k , ranging from 64 to 1. Across both FaaS and CaaS traces, increasing k consistently reduces resource waste at the cost of higher deployment latency. Conversely, smaller values of k maintain larger resource buffers, resulting in lower deployment latency but higher resource waste. The results demonstrate that k provides a smooth and flexible trade-off curve, allowing cloud operators to select operating points that match their latency and efficiency objectives.

6 Discussion

Resource signal faithfulness. In first-party deployments, the cloud platform’s FaaS and CaaS control planes generate resource signals, and tenants typically cannot manipulate them. While a third-party platform could falsely report idle

pre-provisioned instances as active to avoid oversubscription, one billing-based mitigation is to charge a lower rate for idle instances and the normal rate for active ones. AWS Lambda’s provisioned concurrency and Azure Functions’ always-ready instances similarly distinguish charges for provisioned capacity from execution [3, 49].

Tenant fairness. Unipool’s pressure-aware oversubscription can be more aggressive for low-arrival-pressure pools, increasing their risk of eviction or additional deployment latency upon activation. To mitigate this risk, Unipool reserves a minimum amount of resources for every pre-provisioned container. It also disables further oversubscription when a pool’s available capacity falls below a threshold (Section 3.4). It restores each container’s full allocation upon activation (Section 3.6). These safeguards are designed to reduce differences in deployment latency across tenants.

7 Related Work

Resource autoscaling and provisioning. FIRM [56] leverages ML to adaptively provision containers for mitigating SLO violations in microservices. ORION [44] analyzes production FaaS traces to improve function execution and VM provisioning for serverless directed acyclic graph (DAG) applications. RainbowCake [72] and InstaInfer [63] use intelligent resource provisioning to mitigate container cold starts in serverless platforms. UAHS [42] and CAHS [43] predict VM demand uncertainty and accurately optimize VM provisioning strategies. The Kubernetes ecosystem [17, 27, 36] provides representative SOTA frameworks for dynamic container autoscaling based on historical resource usage. However, no prior work has explored the resource inefficiency and optimized resource provisioning through consolidation in cross-layer cloud systems.

Resource oversubscription. Resource oversubscription has been widely explored to improve infrastructure utilization in multi-tenant cloud platforms. Prior work in Database-as-a-Service (DBaaS) environments studies tenant placement and resource allocation under oversubscribed clusters. König et al. [35] investigates tenant placement strategies for oversubscribed DBaaS clusters to improve utilization while maintaining performance isolation. Arora et al. [7] proposes flexible resource allocation mechanisms that dynamically adjust resource assignments among database tenants to better handle workload fluctuations. However, existing approaches primarily focus on oversubscribing running or active resources, whereas Unipool is the first to oversubscribe *pre-provisioned* resources, avoiding interference with runtime resource allocation and execution performance. Runtime resource oversubscription is orthogonal to Unipool and can be integrated with our design to further expand the optimization space.

Resource allocation and placement. Mesos [30], Borg [67], Omega [61], and YARN [66] enable scalable scheduling and efficient resource sharing across shared clusters. Subsequent

systems [19, 26, 55, 64] improve scheduling quality and responsiveness and extend cluster management to region-wide optimization [54]. Resource Central [18], Morpheus [31], and SelfTune [33] leverage workload characterization and adaptive policies, while VM-specific allocators [14, 20, 29, 39] refine deployment decisions under cloud constraints. Other efforts focus on admission control and spare capacity harvesting [60, 77]. In contrast to existing works, Unipool jointly optimizes container management across cloud layers to improve resource efficiency.

Resource harvesting and preemption. Resource harvesting has been widely studied to improve datacenter utilization by exploiting temporarily idle resources. SmartHarvest [70] leverages historical utilization patterns to safely reclaim spare capacity in clusters. MHVM [23] dynamically reclaims unused memory from running virtual machines. BlockFlex [58] enables flexible disaggregation and harvesting of unused storage bandwidth in cloud environments. Recent efforts have also explored harvesting in serverless platforms. Zhang et al. [76] opportunistically utilize harvested VM resources to serve serverless workloads. Other works [73–75] propose harvesting idle function resources in serverless platforms to improve resource efficiency. The cross-layer resource consolidation of Unipool is orthogonal to the above resource harvesting solutions.

8 Conclusion

This paper studies the inefficiencies of PPR in modern cloud platforms and shows that the lack of cross-layer coordination leads to significant resource waste. Through a detailed characterization of production traces, we demonstrate that current PPR management strategies, which operate independently at each layer of the cloud stack, systematically over-provision resources and fail to exploit multiplexing opportunities. Our analysis reveals persistent underutilization caused by hidden idle resources, temporal mismatches between workloads and infrastructure, and the inherent limitations of layer-local optimization policies.

To address these challenges, we present Unipool, a unified framework that consolidates PPR management, enables resource-state visibility across layers, and leverages workload-aware oversubscription to improve efficiency while preserving deployment latency. Our evaluation on mixed FaaS and CaaS production traces through large-scale simulations and experiments on a 25-VM testbed shows that Unipool reduces pre-provisioning resource waste by 92% and lowers average deployment latency by 21% over the strongest baseline.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable feedback. The work of Hanfei Yu and Hao Wang was supported in part by NSF 2534241 and 2523997.

References

- [1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proc. the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
- [2] Ahmed Alquraan, Abdelrahman Baba, Rafael Mendes da Silva, Sameh Elnikety, Paul Batum, Yan Chen, Hamid Henry Safi, Seth Fine, and Samer Al-Kiswany. 2026. DROPS: Managing Serverless Resource Pools in Microsoft Azure Functions. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.
- [3] Amazon Web Services. [n.d.]. AWS Lambda Pricing. <https://aws.amazon.com/lambda/pricing/>. Accessed October 7, 2026.
- [4] Amazon Web Services. 2025. Decrease Latency for Applications with Long Boot Times using Warm Pools - Amazon EC2. <https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-warm-pools.html>
- [5] Lixiang Ao, George Porter, and Geoffrey M Voelker. 2022. FaaSnap: FaaS Made Fast using Snapshot-based VMs. In *Proceedings of the Seventeenth European Conference on Computer Systems*.
- [6] Apache. 2018. Apache OpenWhisk Official Website. <https://openwhisk.apache.org>.
- [7] Pankaj Arora, Surajit Chaudhuri, Sudipto Das, Junfeng Dong, Cyril George, Ajay Kalhan, Arnd Christian König, Willis Lang, Changsong Li, Feng Li, et al. 2023. Flexible Resource Allocation for Relational Database-as-a-Service. *Proceedings of the VLDB Endowment* (2023).
- [8] AWS. 2006. AWS EC2: Secure and Resizable Compute Capacity in the Cloud. <https://aws.amazon.com/ec2/>.
- [9] AWS. 2014. AWS Lambda: Serverless Compute. <https://aws.amazon.com/lambda/>.
- [10] AWS. 2017. Serverless Compute Engine – AWS Fargate. <https://aws.amazon.com/fargate/>.
- [11] AWS. 2018. Managed Kubernetes Service - Amazon EKS. <https://aws.amazon.com/eks/>.
- [12] AWS. 2019. AWS Lambda: Configuring Provisioned Concurrency. <https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html>.
- [13] Bharathan Balaji, Christopher Kakovitch, and Balakrishnan Narayanaswamy. 2021. FirePlace: Placing Firecracker Virtual Machines with Hindsight Imitation. *Proceedings of Machine Learning and Systems (MLSys)* (2021).
- [14] Hugo Barbalho, Patricia Kowaleski, Beibin Li, Luke Marshall, Marco Molinaro, Abhisek Pan, Eli Cortez, Matheus Leao, Harsh Patwari, Zuzu Tang, Larissa Rozales Gonçalves, David Dion, Thomas Moscibroda, and Ishai Menache. 2023. Virtual Machine Allocation with Lifetime Predictions. In *Proceedings of Machine Learning and Systems (MLSys)*.
- [15] Salman A Baset, Long Wang, and Chunqiang Tang. 2012. Towards an Understanding of Oversubscription in Cloud. In *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*.
- [16] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand Container Loading in AWS Lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*.
- [17] Cloud Native Computing Foundation. 2018. Knative. <https://knative.dev/docs/>.
- [18] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. 153–167. <https://doi.org/10.1145/3132747.3132772>
- [19] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC)*.
- [20] David Domingo, Hugo Barbalho, Marco Molinaro, Kuan Liu, Abhisek Pan, David Dion, Thomas Moscibroda, Sudarsun Kannan, and Ishai Menache. 2025. Kamino: Efficient VM Allocation at Scale with Latency-Driven Cache-Aware Scheduling. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- [21] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [22] Ittay Eyal, Flavio Junqueira, and Idit Keidar. 2013. Thinner Clouds with Preallocation. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.
- [23] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. 2022. Memory-Harvesting VMs in Cloud Platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [24] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [25] GCP. 2026. Google Kubernetes Engine (GKE). <https://cloud.google.com/kubernetes-engine>.
- [26] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- [27] Google. 2014. Kubernetes. <https://kubernetes.io/>.
- [28] Google. 2018. gVisor. <https://gvisor.dev/>.
- [29] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. 2020. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- [30] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
- [31] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- [32] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. 2022. Hermod: Principled and Practical Scheduling for Serverless Functions. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC)*.
- [33] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. 2023. Selftune: Tuning Cluster Managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
- [34] Kata Containers Community. 2026. Kata Containers. <https://katacontainers.io/>.
- [35] Arnd Christian König, Yi Shan, Tobias Ziegler, Aarati Kakaraparthi, Willis Lang, Justin Moeller, Ajay Kalhan, and Vivek Narasayya. 2022. Tenant Placement in Over-Subscribed Database-as-a-Service Clusters. *Proceedings of the VLDB Endowment* (2022).
- [36] KServe. [n.d.]. KServe. <https://kserve.github.io/website/docs/intro>. Accessed October 12, 2026.

- [37] Alok Gautam Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Frujeri, Nithish Mahalingam, Pulkit A Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, et al. 2021. Prediction-Based Power Oversubscription in Cloud Platforms. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.
- [38] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-Density Deployment and High-Concurrency Startup in Serverless Computing. In *2022 USENIX Annual Technical Conference (USENIX ATC)*.
- [39] Jianheng Ling, Pratik Worah, Yawen Wang, Yunchuan Kong, Chunlei Wang, Clifford Stein, Diwakar Gupta, Jason Behmer, Logan A. Bush, Prakash Ramanan, Rajesh Kumar, Thomas Chestna, Yajing Liu, Ying Liu, Ye Zhao, Kathryn S. McKinley, Meeyoung Park, and Martin Maas. 2025. Lava: Lifetime-Aware VM Allocation with Learned Distributions and Adaptation to Mispredictions. In *Proceedings of Machine Learning and Systems (MLSys)*.
- [40] Linux Kernel Documentation. 2026. Control Groups v2. <https://docs.kernel.org/admin-guide/cgroup-v2.html>.
- [41] Yubo Liu, Hongbo Li, Mingrui Liu, Rui Jing, Jian Guo, Bo Zhang, Hanjun Guo, Yuxin Ren, and Ning Jia. 2025. FlacIO: Flat and Collective I/O for Container Image Service. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, 87–101. <https://www.usenix.org/conference/fast25/presentation/liu-yubo>
- [42] Chuan Luo, Bo Qiao, Xin Chen, Pu Zhao, Randolph Yao, Hongyu Zhang, Wei Wu, Andrew Zhou, and Qingwei Lin. 2020. Intelligent Virtual Machine Provisioning in Cloud Computing. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI)*. 1495–1502. <https://doi.org/10.24963/ijcai.2020/208>
- [43] Chuan Luo, Bo Qiao, Wenqian Xing, Xin Chen, Pu Zhao, Chao Du, Randolph Yao, Hongyu Zhang, Wei Wu, Shaowei Cai, et al. 2021. Correlation-Aware Heuristic Search for Intelligent Virtual Machine Provisioning in Cloud Systems. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*.
- [44] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh El-nikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and The Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- [45] Ming Mao and Marty Humphrey. 2011. Auto-Scaling to Minimize Cost and Meet application Deadlines in Cloud Workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*.
- [46] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov Test for Goodness of Fit. *J. Amer. Statist. Assoc.* (1951).
- [47] Dirk Merkel et al. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* (2014).
- [48] Microsoft. 2024. Standby Pools for Virtual Machine Scale Sets - Azure Virtual Machine Scale Sets. <https://learn.microsoft.com/en-us/azure/virtual-machine-scale-sets/standby-pools-overview>
- [49] Microsoft Azure. [n.d.]. Azure Functions Pricing. <https://azure.microsoft.com/en-us/pricing/details/functions/>. Accessed October 7, 2026.
- [50] Microsoft Azure. 2012. Azure Virtual Machines. <https://azure.microsoft.com/en-us/products/virtual-machines>.
- [51] Microsoft Azure. 2016. Azure Functions. <https://azure.microsoft.com/en-us/products/functions>.
- [52] Microsoft Azure. 2017. Azure Kubernetes Service (AKS). <https://azure.microsoft.com/en-us/products/kubernetes-service>.
- [53] Microsoft Azure. 2018. Azure Container Instances (ACI). <https://azure.microsoft.com/en-us/products/container-instances>.
- [54] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. 2021. RAS: Continuously Optimized Region-Wide Data-center Resource Allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*.
- [55] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*.
- [56] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX symposium on operating systems design and implementation (OSDI)*.
- [57] Deepak Ravikumar, Alex Yeo, Yiwen Zhu, Aditya Lakra, Harsha Nagulapalli, Santhosh Kumar Ravindran, Steve Suh, Niharika Dutta, Andrew Fogarty, Yoonjae Park, et al. 2024. Intelligent Pooling: Proactive Resource Provisioning in Large-Scale Cloud Service. *The Very Large Data Base Endowment (VLDB)* (2024).
- [58] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghbi, and Jian Huang. 2022. BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*.
- [59] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: Warming Serverless Functions Better with Heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [60] Sultan Mahmud Sajal, Luke Marshall, Beibin Li, Shandan Zhou, Abhisek Pan, Konstantina Mellou, Deepak Narayanan, Timothy Zhu, David Dion, Thomas Moscibroda, and Ishai Menache. 2023. Kerveros: Efficient and Scalable Cloud Admission Control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- [61] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*.
- [62] Mohammad Shahradd, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*.
- [63] Yifan Sui, Hanfei Yu, Yitao Hu, Jianxun Li, and Hao Wang. 2024. Pre-Warming is Not Enough: Accelerating Serverless Inference with Opportunistic Pre-Loading. In *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC)*.
- [64] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- [65] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [66] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin C. Reed, and Eric Baldeschwieler. 2013. Apache

- Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*.
- [67] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*.
 - [68] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 443–457. <https://www.usenix.org/conference/atc21/presentation/wang-ao>
 - [69] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX annual technical conference (USENIX ATC 18)*.
 - [70] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*.
 - [71] Neal Young. 1994. The K-Server Dual and Loose Competitiveness for Paging. *Algorithmica* (1994).
 - [72] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. Rainbow-Cake: 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*.
 - [73] Hanfei Yu, Christian Fontenot, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2023. Libra: Harvesting Idle Resources Safely and Timely in Serverless Clusters. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*.
 - [74] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2022. Accelerating Serverless Computing By Harvesting Idle Resources. In *Proceedings of the ACM Web Conference 2022*.
 - [75] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2024. Freyr+: Harvesting Idle Resources in Serverless Computing via Deep Reinforcement Learning. *IEEE Transactions on Parallel and Distributed Systems* (2024).
 - [76] Yanqi Zhang, Iñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*.
 - [77] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Iñigo Goiri, and Ricardo Bianchini. 2016. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.