

COSMOS: Taming Heterogeneous Resource Demands in Serverless Functions with Complementary Swarming and Monotonic Scale-up

Yunshan Jia, Junyi Shu, *Member, IEEE*, Chiheng Lou, Fangyue Liu, Chenbo Bi, and Xin Jin, *Senior Member, IEEE*

Abstract—Serverless computing is an increasingly popular paradigm built on top of a data center network (DCN), offering scalability and fine-grained billing. However, serverless platforms must efficiently manage resources for a multitude of diverse functions and requests with different inputs. Conventional approaches often ignore differences in resource requirements between functions and individual requests, leading to resource fragmentation, poor cluster utilization, and high overhead from instance initializations.

This paper introduces COSMOS, a novel serverless computing system that improves efficiency across nodes in a cluster by exploiting the complementary resource usage patterns of different functions. Our approach is founded on two key optimizations. First, we route functions with complementary resource requirements to the same node to maximize node utilization, while simultaneously grouping instances of the same function to improve locality and reduce associated overhead. Second, we leverage the emerging capability to resize an instance's resource allocation, allowing our system to reuse existing warm instances for requests with heterogeneous resource demands and minimize instance initializations. Experiments on real-world workloads show that our system improves CPU and memory utilization by up to 46.91% and 45.00%, respectively, and increases overall throughput by up to $2.01\times$ compared to state-of-the-art serverless computing systems.

Index Terms—Serverless computing, Resource-aware scheduling, Heterogeneous resource utilization.

I. INTRODUCTION

Serverless computing is an emerging cloud paradigm built upon the backbone of the Data Center Network (DCN). It abstracts away server management, allowing developers to focus solely on application logic [1]–[3]. In this model, developers deploy programs as cloud functions, and the platform automatically manages the underlying infrastructure, including resource provisioning, auto-scaling and traffic routing. The simplicity and efficiency of serverless computing have led to widespread adoption in domains including web applications [4], data processing pipelines [5], [6], and machine learning workloads [7], [8]. For cloud providers, however, transforming serverless computing into a sustainable business

model necessitates efficient resource utilization across a multi-tenant server fleet.

To achieve high resource utilization, prevailing approaches typically optimize for proxy objectives, such as balancing load [9]–[11] or maximizing the reuse of initialized execution environments (often termed warm instances) [9], [12]–[16]. However, by largely ignoring the diverse resource footprints (e.g., CPU-to-memory ratios) across different functions, these strategies can make sub-optimal routing decisions. If a router sends many memory-intensive functions to a single node, it risks exhausting available memory while letting CPUs remain underutilized. This fragmentation creates stranded resources, preventing the node from processing new requests and thereby lowering overall cluster utilization.

Furthermore, existing resource allocation strategies predominantly operate at the function level, where a static resource capacity is allocated to each function instance [12], [17]–[24]. While prior work recognizes that resource demands can vary across requests [23], [25]–[28], existing solutions often rely on predicting an aggregate demand, selecting a single optimal static configuration [25]–[27], or applying coarse-grained bin-packing [23]. Consequently, these strategies remain fundamentally static and cannot efficiently adapt to fine-grained, input-dependent workload variations at the request level. This one-size-fits-all approach inevitably leads to inappropriate resource allocation, as it either over-provisions resources for light requests or under-provisions for heavy ones.

We confirm these inefficiencies by conducting a comprehensive analysis over a wide range of real-world serverless applications (§II-B). Our analysis shows that: i) The CPU-to-memory ratio (core/GB) varies significantly across different functions, ranging from 0.09 to 10.10, with a coefficient of variation (CV) of 62.43%. This high inter-function heterogeneity leads to significant resource stranding under existing resource allocation strategies that ignore such diversity. ii) At the request level, 54% of functions exhibit substantial variation in resource usage under different inputs, with a CV exceeding 50%. This degree of variability far exceeds what static allocation can efficiently accommodate, forcing over-provisioning to avoid performance degradation and resulting in low resource utilization.

To overcome these inefficiencies, we design COSMOS to proactively model and exploit two complementary sources of heterogeneity: resource-demand diversity across functions and dynamic demand variation within each function. This design is inspired by the serverless practice of decomposing complex, stateful applications into fine-grained, dedicated, stateless functions, which offers an opportunity to characterize resource-

Yunshan Jia, Chiheng Lou, Fangyue Liu, and Xin Jin are with the School of Computer Science, Peking University, Beijing, China. E-mail: {jiayunshan;louchiheng23;lly-wanqiu;xinjinpku}@pku.edu.cn.

Chenbo Bi is with Yuanpei College, Peking University, Beijing, China. E-mail: 2300017783@stu.pku.edu.cn

Junyi Shu is with University of California, Los Angeles, California, USA. E-mail: kyleshu@ucla.edu.

This work was supported in part by the Scientific Research Innovation Capability Support Project for Young Faculty under grant ZYGXQNJSKYCXNLZCXM-II, and US National Science Foundation under grants CNS-2403254, CNS-2330831, and CNS-2106838. (Corresponding author: Junyi Shu.)

use patterns and their input-dependent variation at the function level. At the inter-node level, we introduce complementary swarming, a function request routing strategy that mitigates resource stranding by co-locating functions with opposing resource requirements while preserving locality. At the intra-node level, we design a monotonic scale-up request scheduling policy that reorders the request queue for a function to enforce monotonic resource adjustment that avoids the high overhead of instance restarts. Together, these techniques align resource allocation with modeled demand and significantly improve cluster utilization.

First, to address cluster-wide fragmentation, the complementary swarming routing strategy classifies functions based on their distinct resource profiles (*e.g.*, CPU-intensive vs. memory-intensive). It employs a multidimensional packing algorithm that treats resources as distinct dimensions, intentionally co-locating functions with opposing resource needs on the same physical node. This ensures that a node's full capacity is utilized across all dimensions simultaneously, preventing one resource from becoming a bottleneck while another sits idle.

Second, to address the overhead of handling input-dependent variance, COSMOS utilizes Kubernetes' in-place resizing capability [29], which effectively facilitates unidirectional scaling; while capacity can be seamlessly increased, reducing resources still requires a costly container restart. To minimize container restarts, we introduce monotonic scale-up request scheduling, which organizes function requests into queues sorted by monotonically increasing resource demands. A warm instance processes requests sequentially from a single queue, ensuring that the container only needs to scale up—never down—during a batch. This avoids the latency of cold starts and the overhead of frequent memory reclamation, effectively amortizing the restart cost across a batch of requests.

The key contributions of this paper are as follows:

- We conduct a comprehensive characterization of real-world serverless workloads to determine and quantify the resource inefficiency caused by the heterogeneous resource demands of functions and requests.
- We design COSMOS, a serverless computing system that co-locates functions with complementary resource requirements and dynamically adjusts resources at the request level with complementary swarming for request routing and monotonic scale-up for request scheduling.
- We implement and evaluate COSMOS on real-world workloads, demonstrating up to 46.91% and 45.00% improvements in CPU and memory utilization, respectively, and up to $2.01\times$ the overall throughput of state-of-the-art systems.

II. BACKGROUND AND MOTIVATION

A. Resource Management of Serverless Computing

Serverless computing has emerged as a promising cloud paradigm, allowing developers to focus on application logic by abstracting away infrastructure tasks such as provisioning, scaling, and maintenance. In this deployment model, users simply provide code (functions) with basic resource specifications and triggers. The platform then orchestrates the entire lifecycle—provisioning sandboxed environments, performing

on-demand autoscaling, and managing pay-per-use billing. Consequently, efficient resource management is essential to ensure high performance, maximize utilization, and minimize costs for cloud platforms.

In practice, resource management for a serverless computing cluster is architected along two orthogonal axes: i) vertical sizing, which defines the resource limits for function instances; and ii) horizontal orchestration, which governs autoscaling (instance count) and placement (node selection) of each function.

Instance resource allocation (vertical). Existing serverless computing systems from both industry [17]–[19] and academia [5], [12], [15], [30] allocate a uniform amount of resources for all instances of a given function. In AWS Lambda, a developer specifies a memory size (*e.g.*, 128 MB–10,240 MB), from which the platform derives proportional vCPU and network bandwidth shares [17]. These limits remain fixed throughout the instance lifetime. Similar static per-instance specifications are used by Azure Functions [18] and Google Cloud Run [19]. This approach presumes uniform resource demand across function requests. Consequently, existing optimizations focus on profile-guided configuration tuning [12], [15], [20], [21], [28]—by profiling across different configurations, they quantify the latency–cost trade-off and select the configuration with the minimum cost that satisfies a user-given SLO.

Instance scaling and placement (horizontal). Existing serverless computing systems perform autoscaling and placement based on a static per-function resource requirement [31]–[33]. While autoscalers adjust instance counts in response to load, schedulers employ a resource accounting model where each node tracks allocated versus available capacity. The resource counters are updated only during lifecycle events—decremented upon placement and incremented upon termination—ignoring actual usage during execution. An instance is admitted to a node only if the available capacity meets the function's static requirement (*e.g.*, memory, CPU). For instance autoscaling and placement, recent research has pursued two main optimizations: i) for autoscaling, using prediction to right-size replica counts and prewarm instances [15], [23]; and ii) for placement, improving efficiency through cache-aware [34], heterogeneity-aware [14], and data-affinity strategies [23], [35].

Taken together, the prevailing serverless paradigm relies on modeling each function with a static, uniform resource profile. However, our characterization reveals that this static paradigm conflicts with a dual-layered heterogeneity in resource demands: diverse resource profiles across functions and varying demands within each function. In the next two subsections, we quantify these heterogeneities (§II-B) and then analyze the system-level challenges that they pose to dynamic resource allocation (§II-C).

B. Analysis of Resource Requirements of Functions

Resource Demand Variation Across Functions. The resource demands of serverless functions are highly heterogeneous. We quantify this heterogeneity by analyzing the execution traces of 162 real-world functions from the open-source Huawei dataset [36], calculating the average CPU-to-memory ratio

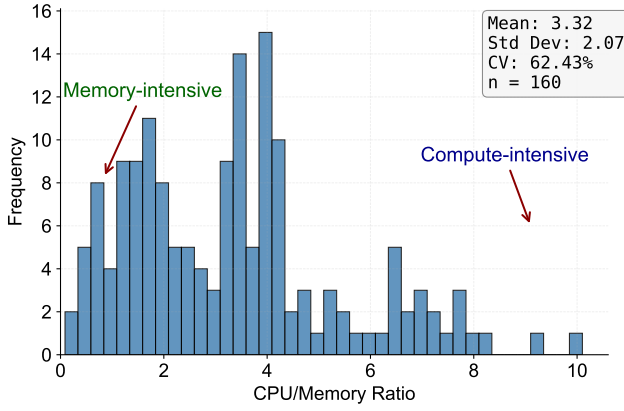


Figure 1. Heterogeneous CPU:memory ratios across functions.

(core/GB) for each. Figure 1 presents the distribution of these ratios across all functions.¹ The distribution is highly dispersed, with ratios ranging from 0.09 to 10.10 and a high CV of 62.43%, indicating that some functions are extremely compute-intensive while others are predominantly memory-intensive. This substantial variation in intrinsic resource profiles implies that a resource allocation strategy that is unaware of such differences is prone to creating resource fragmentation.

Figure 2 further examines how CPU and memory usage co-vary as request inputs change. We randomly select ten functions from the open-source SeBS serverless benchmarking suite [37], covering Web, multimedia, utility, machine-learning, and scientific workloads; their full names and characteristics are listed in Table I. For each input, we measure the CPU time consumed and peak memory usage during task execution, averaged across repeated runs. Each point represents one input, and the red line is a per-function linear fit. The reported R^2 measures the goodness of fit, while CV is the coefficient of variation of the per-input CPU-to-peak-memory ratios. The figure omits graph-mst due to space because it closely matches graph-bfs ($R^2 = 0.996$, $CV=27.9\%$).

Eight of the ten benchmarks exhibit strong input-dependent CPU–memory coupling: their CPU and memory demands increase along a stable function-specific direction, with linear-fit R^2 values between 0.974 and 0.999 and ratio CVs from 5.2% to 29.7%. Thus, although inputs change demand magnitude, the relative CPU-to-peak-memory profile remains broadly stable and does not typically switch between compute- and memory-intensive behavior. This property follows naturally from the fine-grained, single-purpose design of serverless functions: each invocation performs a well-defined task, while complex applications are commonly decomposed into multiple such functions. The remaining two benchmarks, video-processing and uploader, exhibit a streaming pattern: peak memory remains nearly constant while CPU consumption grows with input size.

Opportunity #1: Heterogeneity-aware function placement.

The substantial dispersion in CPU-to-memory ratios across functions creates a significant opportunity to enhance cluster-wide resource efficiency through smarter placement. By leveraging the diverse resource profiles identified in our characteriza-

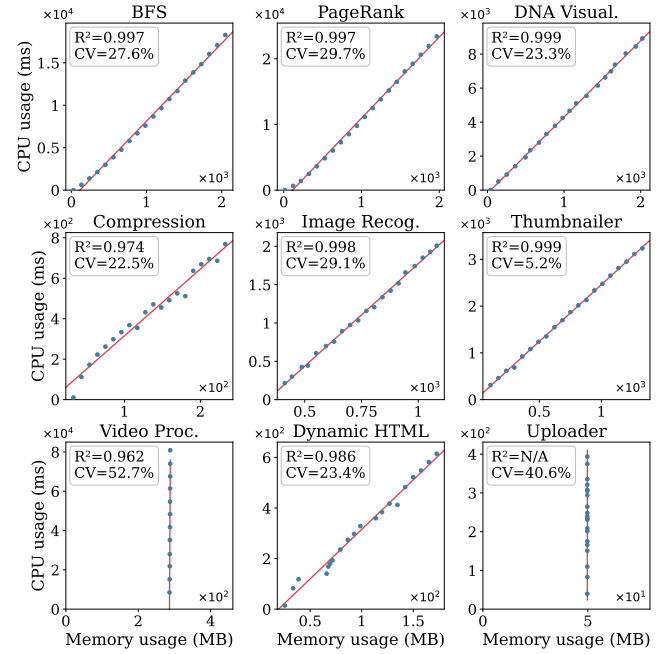


Figure 2. CPU and memory usage across request inputs for serverless functions.

tion, a scheduler can strategically co-locate functions with complementary demands—pairing compute-intensive workloads with memory-intensive ones. This complementary packing balances resource consumption along multiple dimensions at the node level, thereby minimizing fragmentation, maximizing aggregate utilization, and significantly increasing the effective capacity of the server fleet.

Resource Demand Variation Across Requests. Resource demand also varies significantly across different requests of the same function. We compute the coefficient of variation (CV) for the CPU usage of each function across its request trace. The distribution of these CV values is presented in Figure 3. Our analysis shows that variability is the norm rather than the exception: for 54% of functions, the CV exceeds 50%, and an additional 26% have a CV between 15% and 50%. This variation may arise from hardware heterogeneity, communication patterns, and, most commonly, request-input differences (*e.g.*, parameters or data size), as shown in Figure 2. Regardless of the specific cause, this widespread and often substantial variability directly contradicts the assumption of uniform demand underlying static allocation. Consequently, any static resource allocation must provision for the peak demand of such variable functions, inevitably leading to low average utilization.

Opportunity #2: Dynamic in-place runtime resource adjustment.

A key opportunity to address the intra-function variability lies in recent container runtime enhancements that support dynamic, in-place resource adjustment. Specifically, platforms like Kubernetes now allow the resource allocations (CPU and memory) of a running pod to be updated without restarting it, via mechanisms such as the Pod resize subresource and live cgroup updates [29], [38]. This capability enables a container’s assigned resources to be aligned in real time with the actual demand of the incoming request. It thus provides a

¹We compute these statistics using the [Request Table Module, Region 4] portion of the dataset [36] and observe similar distributions in other portions.

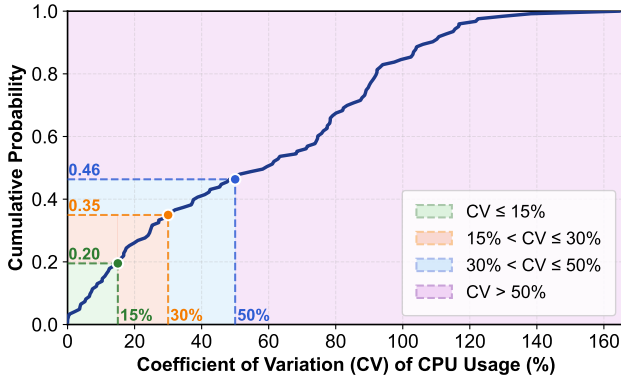


Figure 3. Resource demand variations of serverless functions.

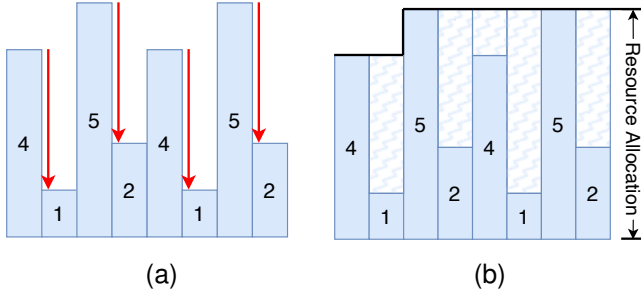


Figure 4. Inefficient scaling strategies under variable workloads. (a) Eager bidirectional resizing with prohibitive reclamation overheads. (b) Increment-only resizing with wasted headroom.

foundational mechanism to move beyond static per-function allocation, toward a fine-grained model where resources can be expanded or contracted per request. It absorbs demand bursts, reduces stranded resources from over-provisioning, and lowers the risk of SLO violation—while preserving the performance benefits of a warm instance.

C. Challenges in Dynamic Resource Allocation for Functions

Challenge #1: Placement under dynamic in-place resizing.

Complementary placement addresses resource fragmentation caused by ignoring heterogeneous resource profiles (*e.g.*, CPU-to-memory ratios). However, dynamic in-place resizing disrupts the straightforward fixed-size bin-packing model: a sandbox's footprint may expand while active, making node feasibility assessments at start/stop times inadequate. A simplistic approach is to evict co-located instances to make space when upscaling faces insufficient capacity. However, this naive strategy risks evicting other co-located function instances with increased resource demand, potentially resulting in eviction amplification: a single significant resize event can force the eviction of multiple smaller co-located instances, thereby hurting cache locality and increasing tail latency. Additionally, fragmentation can occur when resource demand decreases. These problems may lead to low resource utilization and application throughput.

Therefore, an intelligent placement strategy must anticipate and mitigate the disruptive effects of resizing. The goal is twofold: i) to balance CPU and memory usage per node from the outset, preventing resource fragmentation and minimizing the need for disruptive migrations; and ii) to migrate instances in a manner that minimizes overall disruption when necessary. By minimizing instance churn, such a strategy safeguards warm

states (maximizing instance reuse) and maintains high resource utilization even during dynamic resizing.

Challenge #2: High overhead of downsizing. Dynamic in-place scaling is asymmetric: upsizing is swift, but downsizing incurs high costs. Increasing the memory allocation takes effect immediately, whereas reducing it triggers a restart which involves cgroup reclamation, page-cache eviction, and compaction, potentially stalling both runtime and kernel. Since resizing is on the critical path of request handling, this expensive downsizing process can directly degrade both overall throughput and end-to-end latency.

This asymmetry creates a fundamental dilemma for naive instance resizing strategies. If an instance undergoes eager bidirectional resizing for every request, the high downsizing costs identified above incur prohibitive memory reclamation overheads and disrupt page caches and JIT states. Conversely, an increment-only strategy that avoids downsizing inevitably leads to severe over-provisioning, as instances accumulate resources to match their historical peak demand. In either case, the theoretical advantages of dynamic scaling are negated. These inefficiencies are illustrated in Figure 4, which contrasts the reclamation overhead of bidirectional resizing with the wasted headroom of increment-only resizing.

Practical dynamic resource allocation must achieve the following goal: it should adapt resource allocation to varying request-level demands without incurring significant overhead from frequent downsizing. This necessitates a mechanism that can batch or reorder requests to create resource usage patterns that are favorable to increment-only adjustments within short time windows, thereby amortizing reclamation costs and preserving warm instance state.

III. COSMOS OVERVIEW

Figure 5 illustrates the overall architecture of COSMOS, which operates at two levels: inter-node request routing and intra-node request scheduling. At the cluster level, incoming user requests are directed to a request router, which monitors traffic patterns and forwards these statistics, along with function metadata and performance metrics, to the routing plan generator that periodically produces routing plans used to assign requests to nodes. Once assigned to a worker node, requests are entered into queues managed by the worker scheduler. This component maintains separate queues and priorities for different functions, while orchestrating instances within the node to handle request execution and in-place resource scaling. To enable this, COSMOS introduces two key techniques: complementary swarming routing and monotonic scale-up scheduling.

Complementary Swarming Request Routing. COSMOS controls instance placement by establishing a mapping dictating how requests are distributed across the cluster. Using a heuristic algorithm, COSMOS ensures that the collective resource demand of the function mix on each worker node aligns with the node's capacity, thereby balancing CPU and memory usage per node and preventing resource fragmentation. For high-demand functions requiring deployment across multiple nodes, COSMOS employs request-level load splitting that groups requests of similar resource demand onto the same

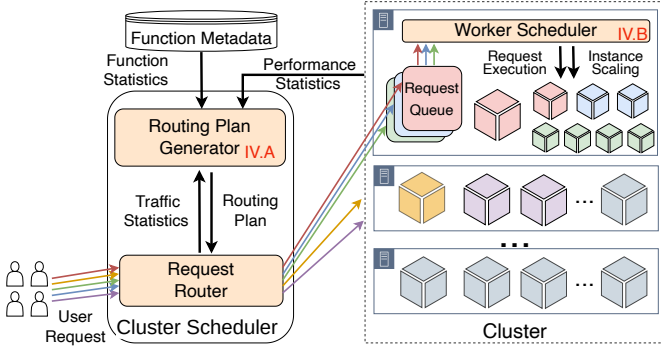


Figure 5. COSMOS overview.

node where possible. This size-aware segregation mitigates the eviction amplification risk under dynamic in-place resizing by preventing uncontrolled, concurrent upsizing across nodes. With controlled optimization step size and partial updates, COSMOS keeps the average algorithm duration under 60 ms.

Monotonic Scale-up Request Scheduling. COSMOS designs its worker-level scheduling algorithm around the restart-free in-place pod scaling feature of Kubernetes. The scheduler maintains dual request queues, both sorted in ascending order of resource demand. This ordering ensures that, within a time window, the resource requirements of requests processed by an instance increase monotonically. Consequently, an instance can be scaled up incrementally to meet the growing demands of successive requests without over-provisioning, thereby increasing resource utilization without incurring restarts. To prevent starvation, COSMOS determines queue priorities by holistically considering request queuing time, cold-start overhead, and resource requirements. By strategically reordering requests under priority constraints, COSMOS effectively boosts resource utilization and reduces the frequency of cold starts.

IV. COSMOS DESIGN

COSMOS employs a two-level hierarchical orchestration approach for serverless functions and requests: complementary swarming request routing at the inter-node level and monotonic scale-up request scheduling at the intra-node level.

A. Complementary Swarming Request Routing

The complementary swarming routing algorithm addresses the inter-node resource fragmentation posed by heterogeneous function resource demands, as quantified in §II-B. It pursues a dual objective: i) maximizing cluster-wide resource utilization by co-locating functions with complementary CPU and memory profiles on the same node, thereby mitigating resource stranding; and ii) enhancing instance reuse by deliberately narrowing the set of distinct functions served per node. To achieve this, the algorithm proceeds iteratively. In each round, it assigns the workload of a function to one or more worker nodes based on the function's resource profile and, when necessary, updates existing workload partitions to improve complementary packing. For functions requiring distribution across multiple nodes, the algorithm performs fine-grained load splitting according to the function's resource demand distribution. The algorithm's outcome is a stable request-to-node mapping that balances complementary packing with demand-aware load distribution.

1) **Modeling Node Capacity and Function Demand:** A precise formulation of the routing problem requires clear representations of both node resource capacity and function resource demand.

Node Resource Capacity. We model a worker node N_j 's resource capacity as a two-dimensional vector $\mathbf{C}_j = (C_j^{\text{cpu}}, C_j^{\text{mem}})$, representing its CPU cores and memory (in GB). Similarly, we denote by $\mathbf{C}_j^{\text{idle}} = (C_j^{\text{idle,cpu}}, C_j^{\text{idle,mem}})$ the vector of idle resources on N_j at any given moment. For simplicity, we focus on CPU and memory as the primary bottleneck resources. Our algorithm can be extended to incorporate other resources (e.g., network bandwidth) by expanding the resource vector. We define the **overall resource utilization** of a node N_j as the sum of its CPU and memory utilization:

$$U(N_j) = \frac{C_j^{\text{cpu}} - C_j^{\text{idle,cpu}}}{C_j^{\text{cpu}}} + \frac{C_j^{\text{mem}} - C_j^{\text{idle,mem}}}{C_j^{\text{mem}}}$$

Here, we simply sum CPU and memory utilization because function demands will be standardized to eliminate weight differences between resources. An optimization step is considered beneficial if it increases $U(N_j)$.

Function Resource Demand. Modeling the resource demand of a function is more nuanced. We characterize it along three orthogonal dimensions:

- 1) **Demand Ratio (r_i):** The intrinsic CPU-to-memory demand ratio required by a single execution of function F_i , where memory demand is represented by the peak memory footprint. The value of r_i quantitatively indicates a function's resource affinity: a relatively small r_i signifies memory-intensive behavior, whereas a relatively large r_i corresponds to compute-intensive behavior. To reduce problem complexity, we assume this ratio remains invariant across requests of the same function. This assumption is consistent with the stable CPU-to-peak-memory profiles observed in Figure 2. We discuss the single-resource-dominant exceptions and the scope of this model in §VII.
- 2) **Demand Volume ($\mathbf{V}_i$):** The aggregate resource demand vector for function F_i within the scheduling window, denoted as $\mathbf{V}_i = (V_i^{\text{cpu}}, V_i^{\text{mem}})$. Here, V_i^{cpu} and V_i^{mem} represent the predicted total CPU (e.g., in core-seconds) and memory (e.g., in GB-seconds) required to process all expected requests of F_i in that window, respectively. Specifically, V_i^{mem} is computed as the peak memory footprint multiplied by execution time, reflecting the memory capacity reserved to avoid OOM. The accurate prediction of V_i is orthogonal to our placement algorithm. COSMOS accepts such a predicted vector as input.
- 3) **Demand Distribution (D_i):** The statistical distribution of per-request resource requirements, derived from historical traces. D_i is crucial for fine-grained load splitting when a function must be spread across multiple nodes.

Together, the tuple $\langle r_i, \mathbf{V}_i, D_i \rangle$ forms a comprehensive demand profile for each function F_i , enabling our algorithm to reason about complementary routing and proportional workload distribution simultaneously.

2) **Complementary Swarming Routing Algorithm:** Algorithm 1 presents the complementary swarming routing workflow.

Algorithm 1 Complementary swarming routing.

Data:

- Pending assignment vector $\mathbf{R} \in [0, 1]^n$.
- Assignment matrix $\mathbf{L} \in [0, 1]^{m \times n}$.

Require:

- Node set $\mathcal{N} = \{N_1, \dots, N_m\}$.
- Function set $\mathcal{F} = \{F_1, \dots, F_n\}$.

Ensure: Optimized routing map $\mathcal{P} : \mathcal{F} \times [0, 1] \rightarrow \mathcal{N}$.

```

1: // Step 1: Initialize state & normalize function demands
2:  $\mathbf{L} \leftarrow \mathbf{0}_{m \times n}$ ,  $\mathbf{R} \leftarrow \mathbf{1}_n$ 
3: for  $F_i \in \mathcal{F}$  do
4:    $\mathbf{V}_i \leftarrow \mathbf{V}_i \times \frac{\sum_{j=1}^m \mathbf{C}_j}{\sum_{i=1}^n \mathbf{V}_i}$   $\triangleright$  Scale demand proportionally
5:
6: // Step 2: Main optimization loop
7: while  $\exists F_i$  s.t.  $R[i] > 0$  do  $\triangleright$  Incomplete mapping
8:    $F_a \leftarrow \text{RandomChoice}(\{F_i \mid R[i] > 0\})$ 
9:   for  $N_j \in \mathcal{N}$  do
10:    // Assignment via free capacity
11:     $\alpha_j^{\text{free}} \leftarrow \min(\frac{C_j^{\text{idle,cpu}}}{V_i^{\text{cpu}}}, \frac{C_j^{\text{idle,mem}}}{V_i^{\text{mem}}})$ 
12:    // Assignment via beneficial swaps
13:     $(\alpha_j^{\text{swap}}, \mathcal{S}_j) \leftarrow \text{SwapCapacity}(F_a, N_j, \mathbf{L}, \mathbf{C}_j^{\text{idle}})$   $\triangleright$ 
     $\mathcal{S}_j$ : internal swap list of  $(F_k, \beta_k)$ 
14:     $\alpha_j^{\text{max}} \leftarrow \min(\alpha_j^{\text{free}} + \alpha_j^{\text{swap}}, R[a])$ 
15:     $\Delta U_j \leftarrow \text{UtilGain}(F_a, \alpha_j^{\text{max}}, \mathcal{S}_j, \mathbf{C}_j)$ 
16:    // Decision & Assignment (ignoring edge cases)
17:    for  $N_j \in \mathcal{N}$ , descending order of  $\Delta U$  do
18:      if  $R[a] \leq 0$  then break
19:       $L[j, a] \leftarrow L[j, a] + \alpha_j^{\text{max}}$ ,  $R[a] \leftarrow R[a] - \alpha_j^{\text{max}}$ 
20:      for  $(F_k, \beta_k)$  in  $\mathcal{S}_j$  do
21:         $L[j, k] \leftarrow L[j, k] - \beta_k$ ,  $R[k] \leftarrow R[k] + \beta_k$ 
22:
23: // Step 3: Generate  $\mathcal{P}$  from  $\mathbf{L}$ 
24: for  $F_i \in \mathcal{F}$  do
25:    $p_{\text{start}} \leftarrow 0.0$   $\triangleright$  Store the workload percentile range
26:   for  $N_j \in \mathcal{N}$ , where  $L[j, i] > 0$ , in random order do
27:     // We use half-open intervals  $[p_s, p_e)$ , except the
     last interval which is closed at 1.
28:      $\mathcal{P}(F_i, [p_{\text{start}}, p_{\text{start}} + L[j, i])) \leftarrow N_j$ 
29:      $p_{\text{start}} \leftarrow p_{\text{start}} + L[j, i]$ 
30: return  $\mathcal{P}$ 

```

It takes as input the set of nodes $\mathcal{N}$ with their capacity vectors $\mathbf{C}_j$, and the set of functions $\mathcal{F}$ with their demand profiles $\langle r_i, \mathbf{V}_i, D_i \rangle$. The algorithm outputs a routing map $\mathcal{P} : \mathcal{F} \times [0, 1] \rightarrow \mathcal{N}$. This mapping determines the request routing rule: when the workload of function F_i is split across multiple nodes, $\mathcal{P}$ assigns each request from F_i to a specific node based on the request's percentile in the function demand distribution D_i . This percentile-based splitting is illustrated in Figure 6.

Step 1: Initialization & Demand Normalization. The algorithm maintains two key data structures initialized in this step: i) a pending assignment vector $\mathbf{R}$, where $R[i]$ denotes the fraction of F_i demand yet to be assigned, initialized to 1 for all functions; and ii) an assignment matrix $\mathbf{L}$, where $L[j, i]$ records the fraction of F_i demand currently assigned to node

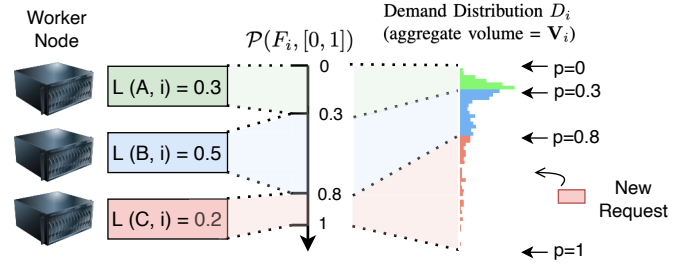


Figure 6. Generating $\mathcal{P}$ from $\mathbf{L}$.

N_j , initialized to all zeros (line 2). Furthermore, we scale the demand vector $\mathbf{V}_i$ of each function proportionally so that the aggregate demand matches the total cluster capacity ($\sum_i \mathbf{V}_i = \sum_j \mathbf{C}_j$). This normalization ensures the subsequent routing operates under a balanced supply-demand premise (Line 3–4).

Step 2: Iterative Optimization Loop. The core of the algorithm is an iterative loop that runs until all function demands are placed ($R[i] = 0, \forall i$, Line 7). In each iteration, it randomly selects a function F_a with pending demand ($R[a] > 0$, Line 8). For each node N_j , the algorithm evaluates two ways to accommodate F_a : i) using the idle node resources, which yields a capacity α_j^{free} computed as the minimum ratio of idle CPU/memory to F_a demand (Line 11); and ii) through beneficial swaps, where the subroutine `SwapCapacity` replaces fractions of existing assignments to host as much of F_a demand as possible, provided that doing so improves the node's utilization $U(N_j)$ (Line 13, detailed in Algorithm 2). `SwapCapacity` returns both the maximum F_a demand fraction α_j^{swap} that can be hosted by function swapping, and an internal swap list $\mathcal{S}_j$ containing pairs (F_k, β_k) , where β_k is the fraction of F_k demand swapped out of N_j . These swaps only modify the assignment matrix during plan computation; they do not directly trigger runtime pod migration or eviction. The total accommodation capacity for F_a on N_j is then $\alpha_j^{\text{max}} = \min(\alpha_j^{\text{free}} + \alpha_j^{\text{swap}}, R[a])$ (Line 14).

After evaluating all nodes, the algorithm sorts them in descending order of the utilization gain ΔU_j (Line 17) and assigns F_a 's pending demand greedily. For each node in this order, it assigns up to α_j^{max} of F_a demand, updates $\mathbf{L}$ and $\mathbf{R}$ (Line 19), and simultaneously processes the internal swaps in $\mathcal{S}_j$: for each function F_k in $\mathcal{S}_j$, the algorithm reduces the swapped fraction β_k in $L[j, k]$ and adds β_k back to F_k pending demand $R[k]$ (Line 21). This process continues until F_a 's demand is fully assigned ($R[a] = 0$). Because the total function demand was scaled to match the total cluster capacity in Step 1, it is guaranteed that F_a demand can eventually be fully accommodated, and the loop will not terminate with $R[a] > 0$. To ensure rapid loop convergence, the algorithm skips nodes whose utilization gain ΔU_j is below a predefined empirical threshold (e.g., 1%).

Step 3: Fine-Grained Routing Plan Generation. After the assignment matrix $\mathbf{L}$ is determined, the algorithm converts the proportional assignments into a precise routing map $\mathcal{P}$ (Line 24–29). This process is illustrated in Figure 6. For each function F_i , the algorithm sequentially allocates contiguous percentile intervals in $[0, 1]$ to each node N_j that hosts F_i

Algorithm 2 SwapCapacity: Compute beneficial swap capacity

Require:

- Node N_j with idle resource vector $\mathbf{C}_j^{\text{idle}}$
- Target function F_a
- Allocation matrix $\mathbf{L}$

Ensure: Swap capacity α^{swap} , internal swap list $\mathcal{S} = [(F_k, \beta_k)]$

- 1: $\alpha^{\text{swap}} \leftarrow 0$, $\mathcal{S} \leftarrow []$, $\{F_k\} \leftarrow \{F_k | L[j, k] > 0\}$
- 2: **if** $C_j^{\text{idle,cpu}} > 0$ **then** $\triangleright$ Determine the dominant idle resource
- 3: // Replace more memory-intensive first
- 4: sort $\{F_k\}$ by r_k **ascending**; sign $\leftarrow +1$
- 5: **else** $\triangleright C_j^{\text{idle,mem}} > 0$
- 6: // Replace more compute-intensive first
- 7: sort $\{F_k\}$ by r_k **descending**; sign $\leftarrow -1$
- 8: **for** each F_k in the sorted order **do**
- 9: **if** sign $\cdot (r_a - r_k) \leq 0$ **then**
- 10: **continue** $\triangleright$ Swap would not improve $U(N_j)$
- 11: Compute max β_k s.t. swapping out β_k of F_k respects $\mathbf{C}_j^{\text{idle}}$
- 12: $\beta_k \leftarrow \min(\beta_k, L[j, k])$
- 13: $\gamma_k \leftarrow \beta_k \cdot (\mathbf{V}_k \odot \mathbf{V}_a) \triangleright \gamma_k = \beta_k \cdot \min\left(\frac{V_k^{\text{cpu}}}{V_a^{\text{cpu}}}, \frac{V_k^{\text{mem}}}{V_a^{\text{mem}}}\right)$
- 14: **if** $\alpha^{\text{swap}} + \gamma_k > R[a]$ **then**
- 15: $\gamma_k \leftarrow R[a] - \alpha^{\text{swap}}$; adjust β_k proportionally
- 16: $\alpha^{\text{swap}} \leftarrow \alpha^{\text{swap}} + \gamma_k$
- 17: $\mathcal{S} \leftarrow \mathcal{S} \cup \{(F_k, \beta_k)\}$
- 18: **if** $\alpha^{\text{swap}} \geq R[a]$ **then break**
- 19: **return** $(\alpha^{\text{swap}}, \mathcal{S})$

(left side of the figure). The length of the interval assigned to N_j is exactly $L[j, i]$. This completes the construction of $\mathcal{P}$, which is the algorithm's final output. In practice, to avoid resource fragmentation that could arise if certain nodes only receive large requests, the intervals are assigned to nodes in a randomized order rather than a fixed sequence; the figure omits this step for simplicity.

Request Routing. At runtime, the interval $[0, 1]$ is concretely mapped to the resource-demand percentiles of F_i 's requests (right side of Figure 6). Specifically, using F_i 's historical demand distribution D_i , we pre-compute the resource-demand boundaries that correspond to each percentile sub-interval. When a new request arrives, its estimated resource demand (derived from input parameters and data size) is located on D_i to obtain its percentile p . The request is then routed to the node N_j for which $\mathcal{P}(F_i, p) = N_j$. For example, the newly arrived request marked in the figure, whose demand percentile falls inside $[0.8, 1]$, would be routed to Node C.

Assignment via beneficial swaps. The SwapCapacity subroutine (Algorithm 2) computes the maximum fraction α_j^{swap} of target function F_a that node N_j can host by proactively identifying beneficial swaps among existing assignments. A swap is beneficial only if it increases $U(N_j)$ by consuming the dominant idle resource: when CPU is idle, F_a replaces more memory-intensive functions; when memory is idle, it

replaces more compute-intensive ones. The subroutine returns the admissible capacity α^{swap} and an internal swap list $\mathcal{S}$.

The subroutine proceeds in four steps. First, it identifies the dominant idle resource—CPU or memory—because only one resource dimension may remain idle after using the node's free capacity in Algorithm 1. Accordingly, SwapCapacity sorts assigned functions by resource-demand ratio r_k : ascending when CPU is idle and descending when memory is idle, allowing the algorithm to swap the most complementary candidates first (lines 2–7).

Second, the algorithm performs a benefit test for each candidate function F_k in the sorted list (line 9). The condition sign $\cdot (r_a - r_k) > 0$ guarantees that swapping a fraction of F_k for F_a will actually increase $U(N_j)$. For example, if CPU is idle (sign = +1), we must have $r_a > r_k$, meaning F_a is more compute-intensive than F_k and will better utilize the spare CPU.

Third, for a candidate that passes the test, the algorithm computes the fraction β_k of F_k that can be swapped out without exceeding the node's idle resources, and limits it by the amount currently assigned to N_j ($L[j, k]$) (lines 11–12). It then translates the released capacity into an equivalent amount of F_a using $\gamma_k = \beta_k \cdot \min\left(\frac{V_k^{\text{cpu}}}{V_a^{\text{cpu}}}, \frac{V_k^{\text{mem}}}{V_a^{\text{mem}}}\right)$ (line 13), ensuring that neither CPU nor memory exceeds the released capacity.

Finally, the algorithm accumulates γ_k until F_a 's pending demand is exhausted or no beneficial candidate remains, then returns α^{swap} and $\mathcal{S}$ to the main loop (lines 14–17).

Low Overhead via Partial Updates. To keep the scheduling overhead low, the routing algorithm employs an incremental (partial) update mechanism. Rather than recomputing the full routing plan from scratch for every scheduling window, the system periodically monitors the latest demand profiles $\langle r_i, \mathbf{V}_i, D_i \rangle$ of all functions. For a function whose aggregate demand $\mathbf{V}_i$ has not increased and whose demand distribution D_i remains statistically stable, the algorithm preserves its current placement. Only functions with significantly changed demand (increased $\mathbf{V}_i$ or altered D_i) are fed into the complementary swarming algorithm to update their assignments. This partial update drastically reduces the number of functions that need to be re-routed in each iteration, thereby reducing plan-level demand relocation and the resulting instance churn while preserving warm states. Furthermore, to prevent sub-optimality from multiple incremental updates, the system performs a global routing-plan recomputation every 10 minutes, recomputing all function assignments from a clean state to ensure long-term placement quality.

B. Monotonic Scale-up Request Scheduling

The monotonic scale-up scheduler runs on each worker node. Its primary objective is to handle the intra-function demand variability identified in §II-B by leveraging in-place vertical scaling, thereby avoiding costly cold starts while maintaining high resource efficiency.

1) *Scheduler Architecture:* As illustrated in Figure 7, the core scheduling structure is organized per function. For each function F_i , the scheduler maintains two monotonic request queues: a current queue and a backup queue. Requests in both queues are ordered by their predicted resource demand in

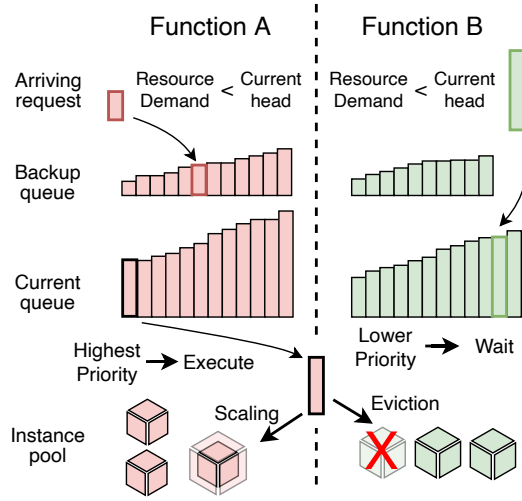


Figure 7. Monotonic scale-up scheduling workflow.

increasing order. This monotonic ordering is the key to enabling efficient in-place scaling: when an instance processes requests sequentially from a single queue, its resource allocation can be increased incrementally without restarts.

Request-insertion rule. When a new request for F_i arrives, its predicted resource demand is compared with the demand of the head (smallest) request in the current queue. If the new demand is smaller, the request is placed into the backup queue; otherwise, it is inserted into the current queue (preserving the monotonic order). The purpose of this rule is not merely to keep each queue sorted, but to guarantee that an instance, while processing requests from a single current queue, encounters a resource demand sequence that is monotonically non-decreasing. This property is essential: if an instance were forced to run a larger request before a smaller one, it would either have to start at the higher capacity (wasting resources) or be restarted after a down-scale (incurring cold-start overhead). The backup queue holds smaller requests until the current queue is processed, after which the queues swap roles.

Instance execution and priority-based selection. A function may have zero or more concurrent warm instances, each dedicated to fetching requests only from that function's current queue. The scheduler assigns a dynamic priority to each function (detailed in §IV-B2). At every scheduling point—triggered by a new request arrival or a request completion—the scheduler selects the function with the highest priority and attempts to process the head request of its current queue.

Resource-assignment policy. To execute the selected request, the scheduler tries the following options in order, aiming to minimize overhead:

- i) **Reuse** an existing instance of the same function whose current resource allocation already meets or slightly exceeds the request's demand, thereby avoiding any scaling operation.
- ii) **In-place scaling** of a warm instance via restart-free vertical scaling (Kubernetes Pod resize API) to accommodate the request, thus preserving warm state.
- iii) **Launch a new instance** (cold start) if no suitable warm instance is available.

If the node lacks sufficient idle resources for options ii) or iii), the scheduler evaluates pod eviction. It selects candidate instances according to a per-sandbox eviction priority:

$$\text{Prio}_{\text{instance}} = \frac{\text{Prio}_{\text{function}}}{N_{\text{instances}}},$$

where $N_{\text{instances}}$ is the number of active instances belonging to the same function. Among instances of the same function, those with larger resource footprints are evicted first. This scheme encourages load spreading (by penalizing functions that already hold many instances) and promotes balanced resource sharing. Pod eviction is performed only if it can actually release enough resources to execute the pending request; otherwise, the scheduler leaves the system unchanged and waits for the next scheduling event.

2) **Priority Design:** The scheduler assigns a dynamic priority to each function F_i to balance three objectives: minimizing cold-start overhead, preventing request starvation, and maximizing the utility of in-place scaling. The priority is computed as a weighted sum of three factors:

$$\text{Priority}(F_i) = \omega_1 \cdot \text{factor}_1 + \omega_2 \cdot \text{factor}_2 + \omega_3 \cdot \text{factor}_3,$$

where the weights $\omega_1, \omega_2 > 0$ and $\omega_3 < 0$ are tunable parameters (set to 0.3, 1.1 and 0.8 respectively in our implementation). The three factors are defined as follows:

- **Cold-start cost factor** (factor_1). Let t_{last} be the request with the largest predicted resource demand in F_i 's current queue, and let $C_{\text{cold}}(F_i)$ denote the estimated cold-start cost of F_i (i.e., the time overhead of launching a new instance). Then

$$\text{factor}_1 = \max\left(\frac{t_{\text{last}}^{\text{mem}}}{C_j^{\text{mem}}}, \frac{t_{\text{last}}^{\text{cpu}}}{C_j^{\text{cpu}}}\right) \cdot C_{\text{cold}}(F_i),$$

where C_j^{mem} and C_j^{cpu} are the memory and CPU capacities of the worker node N_j on which the scheduler is running. This factor increases when the largest pending request occupies a large fraction of the node's resources and the function itself is expensive to cold-start. It encourages the scheduler to keep executing F_i 's requests consecutively, thereby raising the chance of reusing a warm instance.

- **Starvation-avoidance factor** (factor_2). Let W_{max} be the longest waiting time among all pending requests of F_i , and let $\bar{T}(F_i)$ be the average execution duration of requests to F_i . The starvation factor is zero if no request has been waiting excessively; otherwise it grows with the cumulative waiting time and the total work left in the queue:

$$\text{factor}_2 = \begin{cases} 0, & \text{if } W_{\text{max}} < 3 \cdot \bar{T}(F_i), \\ \bar{T}(F_i) \cdot |Q_i| + W_{\text{max}}, & \text{otherwise.} \end{cases}$$

Here $|Q_i|$ is the total number of requests in F_i 's current queue. This mechanism ensures that a function whose requests have been waiting too long receives a priority boost, preventing indefinite starvation.

- **Scaling-efficiency factor** (factor_3). Let t_{first} and t_{last} be the requests with the smallest and largest predicted resource demands in F_i 's current queue, respectively. Under the

assumption that all requests of the same function share the same CPU-to-peak-memory demand ratio (see §IV-A1), the ratio can be computed using either dimension; we use memory for concreteness:

$$\text{factor}_3 = \frac{t_{\text{first}}^{\text{mem}}}{t_{\text{last}}^{\text{mem}}}.$$

A large ratio (close to 1) indicates that the queue's resource-demand range is narrow; consequently, a current instance (if any) requires little or no additional scaling to process all requests in the current queue, resulting in low execution cost. A small ratio (i.e., a wide demand spread) implies that the current instance would need substantial or multiple scaling steps to process the entire queue, making it less beneficial to continue with the current instance.

Finally, to further promote instance reuse, the priority of any function that already possesses one or more warm instances is **doubled**. The combined priority is recomputed at every scheduling event, enabling the scheduler to adapt dynamically to changing workload patterns.

V. IMPLEMENTATION

We implement a prototype of COSMOS with ~3100 LOC in Python, using Kubernetes [39] for instance management and MinIO [40] for function data management.

Instance manager. We use containers as serverless function instances managed through Kubernetes. Each function instance is exposed via a unique Pod IP address, which the scheduler maintains in a pod-IP mapping table. Container resource limits and instance placement are enforced through Kubernetes Pod specifications and node-affinity rules, respectively. In the prototype, the scheduler must manage a large number of instances and frequently interact with each one to perform lifecycle operations—including startup, access, scaling, termination, and health monitoring. To reduce control-plane overhead, our implementation leverages the Python Kubernetes client library in a multi-process and multi-coroutine architecture. Since some benchmarks require reading and writing external data during execution, we additionally deploy a MinIO service within the cluster to provide a unified data management layer for the function system.

Scheduler. The scheduler in the prototype consists of four modules. i) The routing plan generator runs as a periodic cluster-level service. It executes the complementary swarming routing algorithm to compute the optimized request routing plan, with an execution period of five minutes. This interval is configurable: it can be shortened for more volatile workloads or lengthened for stable workloads to reduce the performance disruption caused by load repartitioning. ii) The request router operates as a separate cluster-level process. It assigns each incoming request to a worker node according to the latest routing plan. iii) The worker scheduler runs locally on each worker node. It maintains per-function request queues, applies the monotonic scale-up scheduling policy to determine execution order, and triggers in-place pod scaling when required. iv) The system monitor collects real-time resource usage and per-function metrics (e.g., cold-start time, execution duration, resource demand distribution).

Table I
BENCHMARK SUITE AND EXECUTION CHARACTERISTICS.

Workload	Type	Duration(s)
graph-bfs	Scientific	3.0
graph-mst	Scientific	4.2
graph-pagerank	Scientific	21.5
dna-visualisation	Scientific	7.6
compression	Utilities	1.5
image-recognition	ML	1.6
thumbnailer	Multimedia	5.0
video-processing	Multimedia	11.0
dynamic-html	Webapps	0.15
uploader	Webapps	2.5

VI. EVALUATION

In this section, we first compare COSMOS with Palette [41], FaaSCache [12], and Golgi [42], and use a resource breakdown to explain its performance gains. We then examine behavior under a bursty workload and assess instance churn through demand relocation, pod eviction, and instance lifetime. Finally, we evaluate prediction robustness, generality, the contribution of each design component, and scalability.

A. Evaluation Setup

Benchmarks. We use the ten SeBS benchmarks characterized in §II-B. Table I summarizes their workload categories and execution times, while Figure 2 characterizes their measured task-level resource consumption. The online supplement reports the resource configurations used in our system experiments. These configurations are conservative empirical values selected according to the observed end-to-end runtime behavior of each benchmark and are applied consistently to all systems.

Workload generation. We generate user traffic based on the Azure Functions Dataset [15]. For each benchmark, we randomly sample an invocation trace from the top 20% most frequently invoked functions, which account for most production resource consumption [15]. Their invocation-rate CV averages 55.36% (range: 28.52–97.23%), providing substantial temporal variation. Total invocation counts across the ten benchmarks are equalized within 2%. Because the traces have minute-level granularity, we model intra-minute arrivals as a Poisson process [43]. Offered-load levels are created by proportionally scaling all traces, preserving their temporal patterns across traffic levels. Load is reported as a percentage of cluster CPU capacity, where 100% denotes aggregate profiled CPU demand matching the cluster's allocatable CPU capacity.

We also use the Azure dataset to model within-function demand variation. Shapiro-Wilk [44] and Kolmogorov-Smirnov [45] tests show that the memory-usage distributions of over 95% of functions do not significantly deviate from normality ($p > 0.05$). We therefore randomly select ten functions, fit Gaussian distributions to their memory usage, and adjust each benchmark's inputs and parameters to reproduce the sampled variance. Because the dataset primarily records memory demand, we scale CPU proportionally using each benchmark's resource-allocation ratio reported in the online supplement.

Cluster configuration. Experiments are conducted on Cloud-Lab [46] using a cluster of nine c6620 instances (56 cores total, 2× Intel Xeon Gold 5512U @ 2.1GHz, 128GB memory).

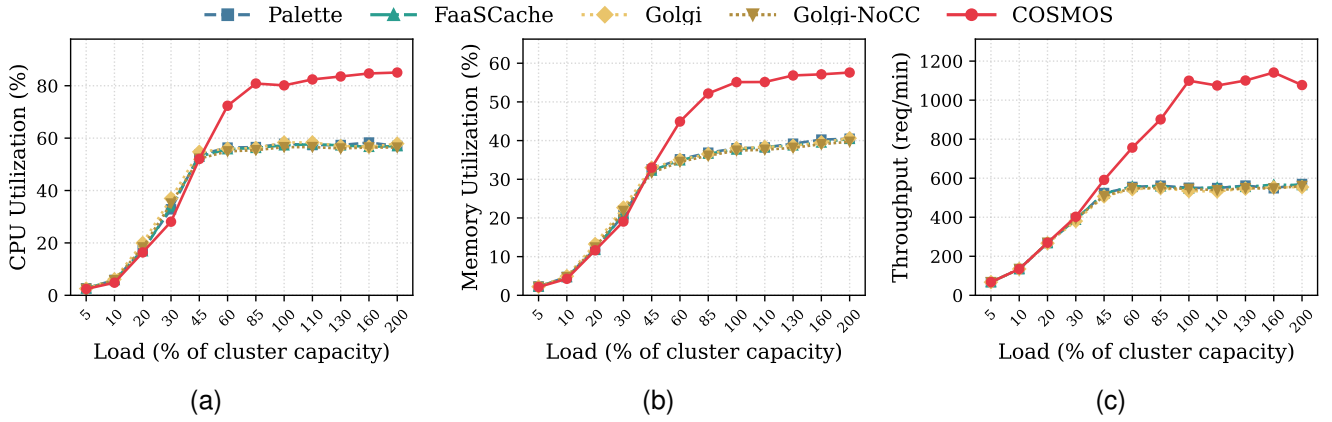


Figure 8. Main-experiment comparison across offered loads: (a) CPU utilization, (b) memory utilization, and (c) task throughput.

Eight nodes serve as worker servers and one as the scheduler and request generator. To emulate the heterogeneous resource availability typical of production clusters, each worker node is allocated only a subset of its total resources. Unless otherwise specified, the available resource proportions across nodes are sampled from a normal distribution with a standard deviation of 0.85.

Baselines. We evaluate COSMOS against three state-of-the-art serverless resource management approaches, which optimize cluster efficiency from complementary perspectives:

- **Palette** [41]: Each benchmark is assigned a unique color, and functions with the same color are co-located on the same server to enhance spatial locality.
- **FaaSCache** [12]: Inspired by caching systems, FaaSCache employs a Greedy-Dual keep-alive strategy that assigns a priority to each kept-alive instance and evicts low-priority instances first under resource pressure.
- **Golgi** [42]: Golgi monitors instance performance using low-level runtime metrics and employs Mondrian Forest models to route requests between overcommitted and non-overcommitted instances. Since COSMOS does not support intra-instance task concurrency, we evaluate two implementations: **Golgi**, which approximates concurrency control in Golgi by adjusting instance resource allocations, and **Golgi-NoCC**, which omits this mechanism. The original Golgi study reports that disabling concurrency control changes performance by at most 14% [42], suggesting that performance monitoring and Mondrian Forest-based routing account for most of the system advantages.

Among these three baselines, only FaaSCache provides a publicly available implementation. To ensure a fair comparison and minimize engineering artifacts, we reimplement all three baselines in our prototype following their respective papers. Other components of the prototype system are kept identical across all evaluated systems.

Metric. We evaluate system performance using CPU/memory utilization and task throughput as primary metrics. To verify that the scheduling algorithm in COSMOS does not lead to task starvation, we also compare the 99th percentile tail latency of task processing. Each experiment runs for 30 minutes,

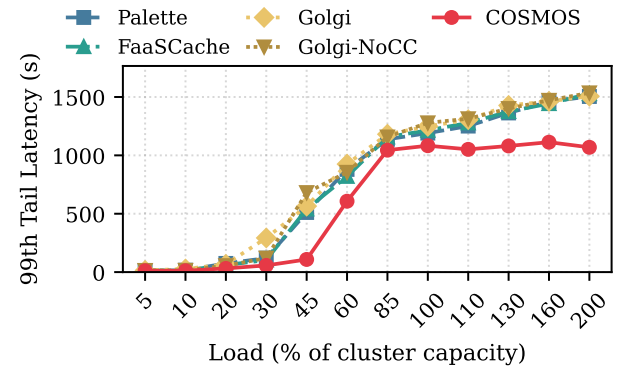


Figure 9. 99th-percentile tail latency across offered loads.

excluding the first five minutes to eliminate cold-start effects. Results are averaged over three runs.

B. Benefits of COSMOS

Figures 8 and 9 compare COSMOS with the baselines across offered loads from 5% to 200% of cluster capacity. Unless otherwise stated, all reported differences are relative to the best-performing baseline for the corresponding metric at each load. At light loads (5%–30%), all systems achieve nearly identical throughput, with differences within 5.82%, because resources are abundant and scheduling efficiency is not yet a bottleneck. The baselines also perform similarly, except that Golgi generally incurs higher tail latency, suggesting that its relatively aggressive overcommitment involves a tail-latency trade-off.

The advantage of COSMOS emerges as the offered load approaches cluster capacity. At 45% load, COSMOS improves throughput by 12.83% and reduces the 99th-percentile latency by 78.77%. From 60% to 200% load, all baselines reach a similar throughput ceiling. In contrast, COSMOS improves CPU utilization by 28.34%–46.91%, memory utilization by 27.76%–45.00%, and throughput by 35.63%–101.31%. Over the same range, COSMOS reduces the 99th-percentile latency by 8.19%–28.94%, showing that its throughput gains do not compromise tail latency.

These gains stem from two complementary mechanisms. Complementary swarming narrows each node's function set to improve reuse while matching resource profiles to node capacity to reduce fragmentation. Monotonic scale-up further reduces over-provisioning and cold starts, sustaining higher utilization and throughput as load rises.

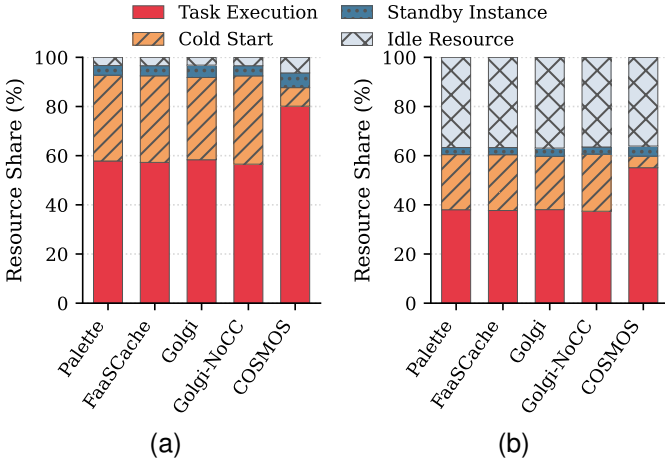


Figure 10. Resource usage breakdown of COSMOS and the baselines under the default configuration at 100% offered load: (a) CPU and (b) memory.

C. Performance Analysis

This subsection further analyzes the sources of COSMOS's performance gains through a resource-consumption breakdown, the system behavior under bursty workloads, the performance impact of routing-plan updates, and the practical effectiveness of monotonic scale-up, as well as its robustness to demand-prediction errors.

Detailed resource consumption. We first examine how COSMOS changes resource consumption during task processing. Figure 10 breaks down the average CPU and memory capacity of COSMOS and the baselines into the following four categories:

- **Task Execution:** resources allocated for processing tasks;
- **Cold Start:** resources consumed for instance initialization;
- **Standby Instance:** resources held by idle instances;
- **Idle Resource:** available but unallocated resources.

At 100% offered load, COSMOS increases the task-execution share of CPU and memory by 37.34% and 44.84%, respectively, over the best-performing baseline. Meanwhile, it reduces the corresponding cold-start shares by 77.46% and 78.34%, and increases the standby-instance shares by 19.01% and 22.38%. Despite processing substantially more tasks, COSMOS retains 85.77% more idle CPU than the baseline with the largest idle CPU share, while the idle-memory share of COSMOS remains comparable to the baselines, differing by at most 2.24%.

The resource utilization advantage of COSMOS stems from the reduction in cold starts. This is achieved through two strategies. First, the complementary swarming routing concentrates a narrower set of functions on each node, which increases instance reuse opportunities. Second, the monotonic scale-up scheduling policy reorders requests and dynamically adjusts instance resources, enabling instances to serve varying demands throughout their lifecycle without repeated cold starts or static over-provisioning.

The resources conserved from fewer cold starts not only increase the share for task execution but also expand the pool of idle resources and standby instances, facilitating container reuse and in-place scaling.

Handling bursty workloads. Figure 11 shows the minute-level cold-start rate and 99th-percentile tail latency after excluding

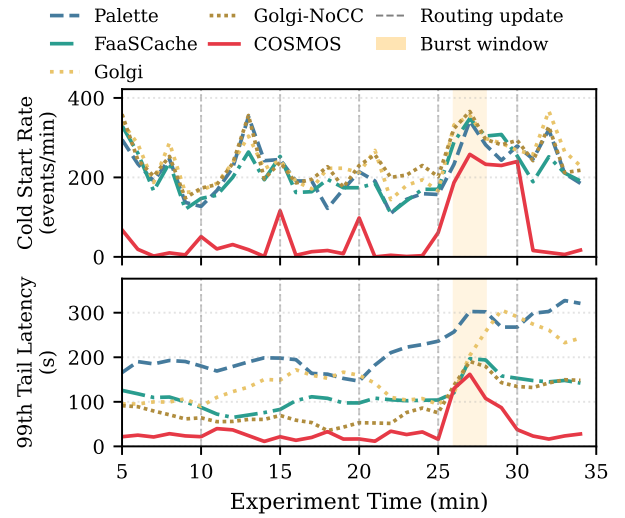


Figure 11. Minute-level cold-start rate and 99th-percentile tail latency under bursty load and routing-plan updates. Dashed vertical lines mark COSMOS routing-plan updates, and shaded regions indicate detected burst windows.

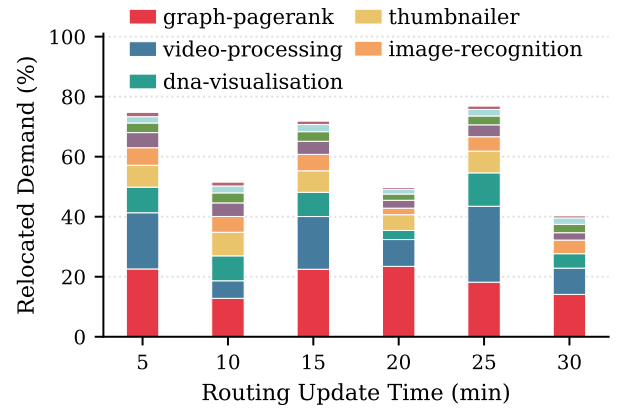


Figure 12. Per-function demand relocation between consecutive COSMOS routing plans for Figure 11. The legend identifies the five functions with the largest average relocated shares.

the first five minutes of system warm-up. We present results at 45% offered load because the baselines already exhibit substantial request backlogs at 100% load.

Between minutes 26 and 28, the invocation rate of **graph-pagerank** rises by about $5\times$. This burst is particularly challenging because **graph-pagerank** is among the most resource-demanding benchmarks. As shown in the figure, the burst causes sharp increases in both cold starts and tail latency for all systems. Nevertheless, COSMOS incurs fewer cold starts and a smaller latency spike than the baselines, and its tail latency quickly returns toward its pre-burst level. In contrast, the baselines retain elevated tail latency and recover more slowly.

This difference arises because complementary swarming confines individual function demand fluctuations to a small subset of servers, while the resulting resource-complementary load mix mitigates intra-server resource contention. Together, these properties limit both the cluster-wide impact of a burst and its performance interference with other functions. The baselines lack this isolation, allowing burst-induced contention and request backlogs to affect more servers and prolong latency recovery. For sustained workload changes, COSMOS revises the cluster load partition at the next routing-plan update to realign resources with demand.

Table II
POD EVICTION RATE AMONG COSMOS AND BASELINES FOR FIGURE 11.

System	Palette	FaaSCache	Golgi	Golgi-NoCC	COSMOS
Average rate (events/min)	5.433	4.500	8.967	3.633	1.433

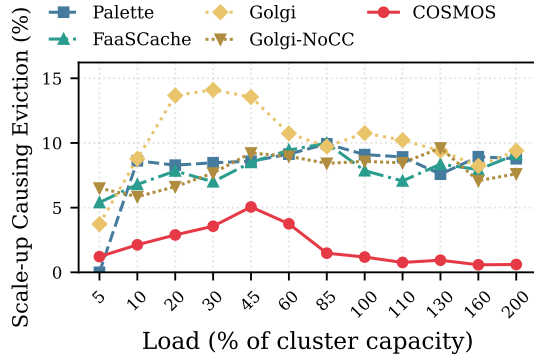


Figure 13. Fraction of scale-up events that cause pod eviction across offered loads.

Impact of routing-plan updates. Figure 12 and Table II further examine the routing-plan updates in the experiment shown in Figure 11. The figure decomposes the net per-function demand relocation between six pairs of consecutive routing plans, while the table reports the observed runtime pod eviction rates. This plan-level metric captures the aggregate effect of beneficial swaps on function-load placement, rather than counting internal swap operations, which are never deployed individually. The even-numbered updates relocate only 40.4%–51.7% of total demand, substantially less than the 72.0%–76.9% relocated by the odd-numbered updates, demonstrating the effect of partial updates.

The overall relocated share remains relatively high because COSMOS deliberately adopts an aggressive reallocation policy: upon detecting an increase in a function's load, it reallocates that function's full demand rather than only the increment. This design prioritizes concentrating each function on fewer servers, as function concentration has a greater performance impact than minimizing the magnitude of each load adjustment. Routing-plan changes are applied lazily as subsequent requests arrive rather than by eagerly migrating all affected instances, so the plan-level relocation overstates the workload immediately affected at runtime. As shown in Figure 11, COSMOS does not exhibit routing-update-induced tail-latency spikes outside the burst window. Moreover, COSMOS incurs fewer cold starts than the baselines, and its pod-eviction rate is 1.433 events/min, 60.6% lower than the best baseline in Table II. These results show that routing-plan relocation does not increase observable instance churn.

Runtime impact of monotonic scale-up. Figure 13 reports the fraction of scale-up events that trigger pod eviction across offered loads, while Figure 14 compares the instance-lifetime distributions of four representative functions. Together with the overall pod-eviction rate and cold-start results, these metrics evaluate whether the eviction-amplification risk of in-place scaling translates into increased instance churn.

For COSMOS, the fraction of scale-ups causing eviction

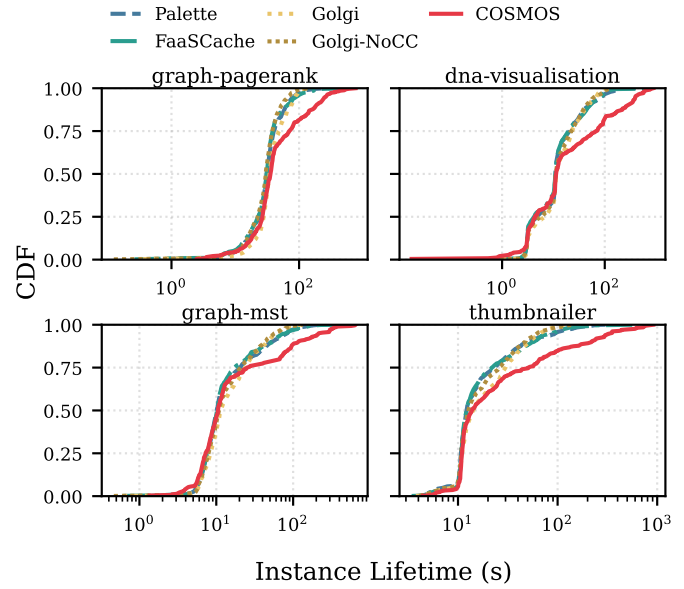


Figure 14. CDFs of instance lifetimes for representative functions in Figure 11.

first increases and then decreases with load. At light loads, eviction events initially grow faster than scale-up events. As load increases further, however, scale-ups become much more frequent and are predominantly absorbed by available node resources. From 45% to 100% load, the number of scale-up events increases by 6.34 $\times$, whereas the number causing eviction increases by only 1.48 $\times$; consequently, their fraction falls from 5.05% to 1.18%. Across COSMOS and all baselines, fewer than 15% of scale-up events cause an eviction, indicating that in-place scale-up usually preserves warm instances rather than triggering additional cold starts.

Consistent with this result, the instance-lifetime CDFs in Figure 14 shift toward longer lifetimes under COSMOS. We observe the same trend for the remaining six functions. These results show that monotonic scale-up does not increase instance churn.

Impact of demand-prediction errors. Figure 15 compares COSMOS without injected prediction errors against variants with $\pm 10\%$ and $\pm 20\%$ random errors and $\pm 15\%$ systematic biases across offered loads. For a $\pm x\%$ random-error setting, we independently multiply each function's predicted load at every routing update by $1 + \epsilon$, where ϵ is uniformly sampled from $[-x, x]$. For a $\pm y\%$ systematic-bias setting, every predicted function load is multiplied by the fixed factor $1 + y$ or $1 - y$, respectively. For readability, we show only Palette and FaaSCache as representative baselines. When offered load remains within cluster capacity, the impact of injected errors remains bounded because available resource slack can absorb moderate allocation inaccuracies: relative to the unperturbed setting, throughput deviates by at most 9.79%, while CPU and memory utilization deviate by at most 13.70% and 7.61%, respectively. Once load exceeds cluster capacity, inaccurate predictions can create imbalanced allocations, producing less stable performance across load levels; independently sampled errors further amplify this variability across functions and routing-plan updates. Nevertheless, all prediction-error variants retain a clear and consistent advantage over the baselines,

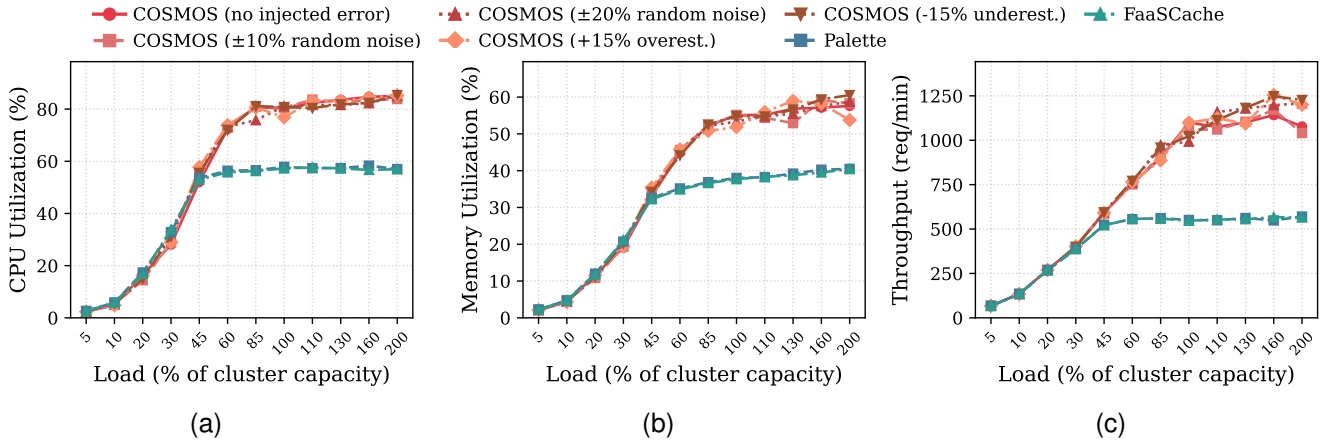


Figure 15. Robustness under demand-prediction errors across offered loads: (a) CPU utilization, (b) memory utilization, and (c) task throughput.

Table III
GENERALITY PERFORMANCE AT 100% OFFERED LOAD.

Setting	CPU util. (%) Best/COSMOS (gain)	Throughput (req/min) Best/COSMOS (gain)
<i>Default configuration</i>		
Main experiment	58.3/80.1 (+37.34%)	551/1100 (+99.56%)
<i>Resource distribution</i>		
Uniform	58.6/80.6 (+37.63%)	808/1514 (+87.30%)
Zipf	56.0/80.1 (+43.04%)	785/1493 (+90.10%)
<i>Number of functions</i>		
5 functions	74.4/88.7 (+19.15%)	290/494 (+70.44%)
20 functions	54.5/77.8 (+42.73%)	545/1063 (+94.97%)
<i>Workload skew</i>		
80/20 popularity	68.3/80.8 (+18.31%)	564/666 (+18.00%)

indicating that COSMOS's overall performance gains are robust to imperfect demand estimates.

D. Generality

We evaluate COSMOS across node-capacity distributions, function-set sizes, and function-popularity distributions. Table III first reports the default configuration from the main experiment and then summarizes three generality dimensions at 100% offered load. Each entry gives the best baseline and COSMOS absolute values, followed by COSMOS's relative gain. To save space, subsequent experiments omit Golgi-NoCC.

Cluster resource distributions have limited impact on COSMOS's gains, whereas reduced workload diversity, caused by fewer functions or skewed popularity, lowers the optimization headroom because fewer workload types naturally increase instance reuse even for the baselines. Nevertheless, COSMOS improves both CPU utilization and throughput in all settings. Since real platforms host far more popular functions than our testbed, these experiments demonstrate generality rather than suggest that COSMOS is unsuitable for skewed real-world workloads. The online supplement provides the complete load-sweep figures, scenario definitions, and detailed analysis.

E. Effectiveness of COSMOS

In this section, we investigate the impact of the techniques employed by COSMOS. We evaluate COSMOS using the following two simplified variants:

- **Routing***: This variant uses only the complementary swarming request routing strategy, while tasks within each node are scheduled using a simple FIFO policy.

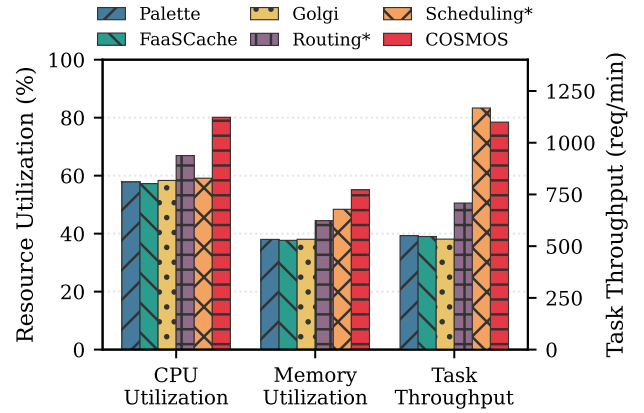


Figure 16. Performance comparison among COSMOS, the baselines, and simplified variants.

- **Scheduling***: This variant uses only the monotonic scale-up scheduling policy within each node, while user requests are randomly routed across worker nodes.

Figure 16 shows the performance comparison between COSMOS and simplified variants. We also provide the evaluation results of the baselines for reference.

As shown, Routing* performs between the baselines and COSMOS. At 100% offered load, it improves CPU utilization, memory utilization, and task throughput over the best-performing baseline by 14.71%, 16.80%, and 28.57%, respectively. However, it remains below COSMOS by 16.48%, 19.36%, and 35.57% on the same metrics. These results show that complementary swarming alone improves resource efficiency and throughput, but requires monotonic scale-up for fine-grained intra-node resource management.

Scheduling* exhibits a different trade-off. Its CPU and memory utilization are 26.23% and 12.19% lower than COSMOS, respectively, while its throughput is 6.21% higher. This result suggests that, without complementary routing, the node-local scheduler favors immediately schedulable requests and thereby increases the completed-request count, but does not align each node's workload with its multidimensional resource capacity. In contrast, COSMOS uses complementary routing to admit a more resource-efficient workload mix, substantially improving CPU and memory utilization at the cost of slightly lower throughput in this setting.

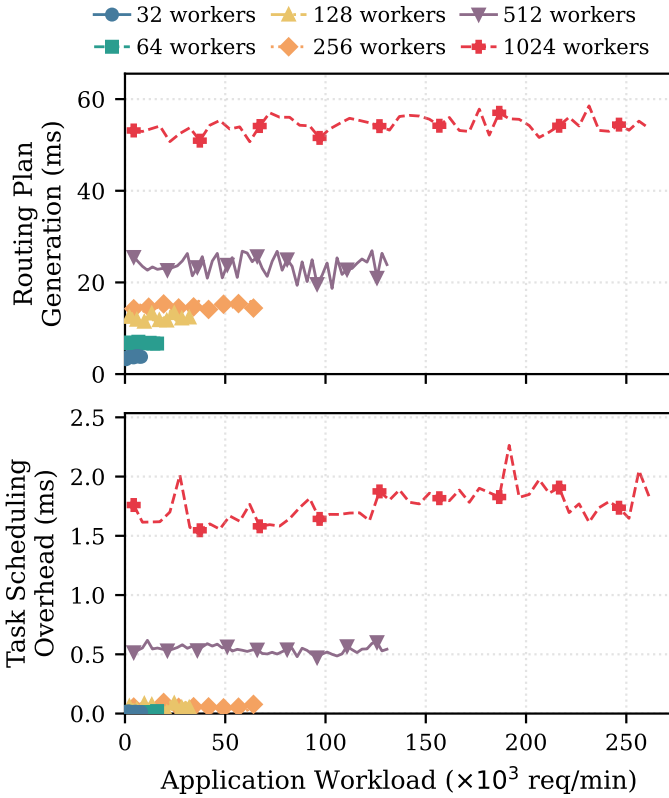


Figure 17. Routing-plan generation and task-scheduling overhead under different cluster sizes and application workloads.

F. Scalability

Finally, we evaluate the scalability of COSMOS. In a 16-worker cluster experiment, COSMOS improves CPU utilization, memory utilization, and task throughput over the best-performing baseline by 35.78%, 39.40%, and 67.43%, respectively, at 100% load; the online supplement reports the complete results. We then evaluate scheduler overhead on simulated clusters of up to 1024 workers.

System overhead. We evaluate the overhead of the COSMOS scheduler under varying cluster sizes and user traffic. To do so, we simulate scheduler load for clusters ranging from 32 to 1024 worker nodes, scaling function load proportionally with cluster size and sweeping each cluster up to 150% offered load. The experimental results are presented in Figure 17.

The upper panel of Figure 17 shows the average duration for COSMOS to generate cluster-wide request routing plans. It can be observed that the generation duration increases gradually with cluster scale but remains largely unaffected by invocation frequency. This is because COSMOS scales the overall load proportionally to the cluster size before computing the routing plan. Even with 1024 worker nodes and an invocation rate above 260×10^3 requests per minute, the latency stays below 60 ms. This result indicates that the scheduler can generate routing plans promptly, even in large-scale clusters, demonstrating the practical scalability of COSMOS.

The lower panel of Figure 17 shows the scheduling overhead for individual tasks under different scenarios. It should be noted that routing plan generation is performed by a separate background process within the scheduler. The main scheduling

process only receives the resulting routing plan and assigns tasks accordingly. Therefore, only the scheduling overhead contributes to the additional latency in the end-to-end task execution path. As shown, even with 1024 worker nodes under heavy load, the extra scheduling delay added by COSMOS remains below 2.5 ms. This overhead is negligible for the vast majority of FaaS scenarios, confirming that the scheduling logic of COSMOS remains computationally lightweight at large cluster scales.

VII. DISCUSSION

Scope of the resource-profile model. Serverless computing commonly decomposes complex, stateful applications into fine-grained, dedicated, stateless functions that scale independently, with state externalized to storage services. Because each function performs a well-defined task, request inputs often change demand magnitude rather than its qualitative CPU-to-memory profile, as illustrated by eight of the ten functions in Figure 2. Motivated by this structure, COSMOS models each function with a stable CPU-to-memory demand ratio and a variable demand magnitude.

Single-resource-dominant functions, such as video-processing and uploader in Figure 2, can still be represented by a conservative fixed resource ratio. This may over-allocate the less-variable resource but preserves normal system operation. Specialized support would require extending the demand model and SwapCapacity routine to allow the ratio to vary with request size.

The current model does not directly support a complex function whose resource profile changes qualitatively across requests. Such a function could be divided into virtual functions by request type or refactored into dedicated stages, each with a stable profile. We leave direct support for such functions to future work.

Vertical scaling versus size-specific pools. A natural question is why predictive horizontal scaling with size-specific prewarmed pools cannot replace monotonic vertical scaling. Such pools face an inherent granularity trade-off. Coarse classes use static sizes that cannot closely match every request, retaining within-class over-provisioning. Fine classes fragment prewarmed capacity into many small pools; a burst in one class can cause queuing or cold starts even while warm instances in other classes remain idle. Monotonic vertical scaling avoids this trade-off by sharing warm instances across request sizes without predefined class boundaries. Size-specific pools remain effective when a function has only a few stable request types with predictable per-class load; otherwise, vertical scaling better accommodates heterogeneous and bursty demands.

Algorithm inputs and demand estimation. As shown in §IV-A, COSMOS requires cluster node capacities and function resource-demand information when constructing routing and scheduling decisions. Node capacities are normally available to the scheduler, whereas function resource-demand modeling requires measurements from historical executions. COSMOS also consumes predictions of future per-function load and request-level resource demand. These predictions can be supplied by any demand forecasting algorithm, but the algorithms

themselves fall outside the scope of this paper. In the prototype system, we use an FFT-based prediction algorithm to forecast future load and employ first- to second-order linear fitting functions of request scale to predict per-request resource demands. These simple strategies have proven sufficient to meet the system requirements. Furthermore, COSMOS exhibits a certain tolerance towards inaccuracies in future load prediction, as shown in Figure 15.

VIII. RELATED WORK

Serverless runtime resource configuration. Optimizing per-instance resource allocation for serverless functions is a well-explored area of research. A prevalent method is profile-guided static tuning [15], [30], involving benchmarking each function across various resource setups (e.g., memory sizes, CPU cores) to establish a latency-vs-cost curve. The least costly configuration that fulfills the service level objectives (SLOs) is chosen for deployment. Other works [5], [20]–[24] go further to optimize resource allocation for serverless workflows by modeling the relationship between E2E latency and individual function resources. Cloud providers' tools (e.g., AWS' power tuning tool [28]) employ this technique to assist users in optimizing memory allocations. Unfortunately, all of them assume a static and uniform resource allocation per function, ignoring the significant per-request resource usage variations. Such static allocation can lead to over-provisioning for lightweight requests, resulting in suboptimal utilization. By enabling dynamic resizing of serverless function resources, our work addresses this limitation, allowing more efficient resource usage.

Instance Orchestration and Placement. This line of work focuses on managing the lifecycle, quantity, and location of function instances, typically under a static per-instance resource model. A primary concern is optimizing the instance count to balance cost and cold-start latency. Prior work keeps instances alive [15], prewarms them based on predicted demand [14], [23], or scales them using runtime metrics [24], [31]–[33]. To improve placement efficiency, recent work considers several dimensions: i) Cache awareness, which reuses instance states (e.g., through copy-on-write cloning) to reduce startup overhead [34], [47]; ii) Cost-optimized placement, which matches functions to specialized or lower-cost hardware (e.g., spot instances, GPUs) to reduce deployment expenses [14], [48]; and iii) Data affinity, which co-locates dependent tasks in serverless workflows to minimize data transfer, as seen in systems like Wukong [35], FaaSFlow [49], ORION [23], and Ditto [20]. However, a fundamental premise shared by these approaches is the assignment of a static, uniform resource capacity to each function instance. In contrast, COSMOS models demand to coordinate resource allocation and function-load partitioning, mitigating fragmentation and burst-induced performance variation. Golgi [42] maintains differently sized, overcommitted instances but routes requests solely by latency predictions and online performance statistics. This reactive design is sensitive to transient variations and forgoes resource-profile-aware instance reuse.

Request and Task Scheduling. For request and task scheduling, prior work has focused on improving scheduling efficiency, load balancing, and resource cost optimization. For instance, Hermod uses cost, load, and locality awareness to optimize consolidation and reduce cold starts [9]. XFaaS defers delay-tolerant functions and dispatches calls globally to handle load spikes [50]. Atoll employs a tiered architecture with semi-global schedulers and sandbox-aware routing [51]. Kaffes et al. advocate for a centralized, core-granular scheduler to eliminate queue imbalance [52]. Golgi minimizes costs through carefully managed overcommitment based on historical usage [42]. However, these scheduling policies largely ignore per-request resource demand variations. Our monotonic scale-up scheduling policy reorders function requests to ensure resource demands increase monotonically, thereby amortizing cold start overhead and avoiding over-provisioning.

IX. CONCLUSION

This paper presents COSMOS, which addresses heterogeneous function demands through complementary swarming routing and monotonic scale-up scheduling. At the cluster level, COSMOS co-locates complementary functions to improve server utilization while narrowing each node's function set to increase instance reuse. At the node level, it aligns request ordering with in-place pod scaling, adapting allocations to request demands and reducing cold starts without over-provisioning. Our evaluation shows higher resource utilization and throughput than the baselines, with negligible overhead and scalable performance.

ACKNOWLEDGMENTS

This research is supported by the Huawei Peking OS Innovation Lab.

REFERENCES

- [1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, 2019.
- [2] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A Berkeley view on serverless computing," Tech. Rep., 2019.
- [3] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: the next phase of cloud computing," *Communications of the ACM*, 2021.
- [4] Microsoft, "Serverless web application - azure architecture center," <https://learn.microsoft.com/en-us/azure/architecture/web-apps/serverless/architectures/web-app>, 2025, accessed: 2025-10-10.
- [5] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *16th USENIX symposium on networked systems design and implementation (NSDI 19)*, 2019, pp. 193–206.
- [6] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *USENIX OSDI*, 2018.
- [7] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *The International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2020.
- [8] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu, "Dorylus: Affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads," in *USENIX OSDI*, 2021.

- [9] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: principled and practical scheduling for serverless functions," in *ACM Symposium on Cloud Computing*, 2022.
- [10] Knative, "Knative," <https://knative.dev/>, 2025, accessed: 2025-10-10.
- [11] O. Ltd., "Openfaas," <http://openfaas.com/>, 2025, accessed: 2025-10-10.
- [12] 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*, ser. ASPLOS '21, New York, NY, USA, 2021, pp. 386–400.
- [13] 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 *ACM ASPLOS*, 2024.
- [14] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: warming serverless functions better with heterogeneity," in *Proceedings of the 2022 ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Lausanne, Switzerland, 2022, pp. 753–767.
- [15] M. Shahradd, R. Fonseca, I. Goiri, G. Chaudhry, P. Batur, 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," in *Proceedings of the 2020 USENIX Annual Technical Conference*. California, USA, 2020, pp. 205–218.
- [16] T. A. S. Foundation, "Apache openwhisk," <https://openwhisk.apache.org/>, 2025, accessed: 2025-10-10.
- [17] A. W. Services, "Configure lambda function memory," <https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html>, 2025, accessed: 2025-10-10.
- [18] Microsoft, "Azure functions hosting options," <https://learn.microsoft.com/en-us/azure/functions/functions-scale>, 2025, accessed: 2025-10-10.
- [19] Google, "Configure memory limits for services," <https://cloud.google.com/run/docs/configuring/services/memory-limits>, 2025, accessed: 2025-10-10.
- [20] C. Jin, Z. Zhang, X. Xiang, S. Zou, G. Huang, X. Liu, and X. Jin, "Ditto: Efficient serverless analytics with elastic parallelism," in *Proceedings of the ACM SIGCOMM 2023 Conference*. New York, NY, USA, 2023, pp. 406–419.
- [21] Z. Zhang, C. Jin, and X. Jin, "Jolteon: Unleashing the promise of serverless for serverless workflows," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 167–183.
- [22] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: NIMBLE task scheduling for serverless analytics," in *Proceedings of the 2021 USENIX Symposium on Networked Systems Design and Implementation*. Boston, MA, USA, 2021, pp. 653–669.
- [23] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chatterji, and S. Bagchi, "ORION and the three rights: Sizing, bundling, and prewarming for serverless dags," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*. Carlsbad, CA, USA, 2022, pp. 303–320.
- [24] Z. Zhou, Y. Zhang, and C. Delimitrou, "AQUATOPE: qos-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proceedings of the 2023 ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. BC, Canada, 2023, pp. 1–14.
- [25] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "CherryPick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 469–482.
- [26] A. Klimovic, H. Litz, and C. Kozyrakis, "Selecta: Heterogeneous cloud storage configuration for data analytics," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 759–773.
- [27] A. Mahgoub, A. M. Medoff, R. Kumar, S. Mitra, A. Klimovic, S. Chatterji, and S. Bagchi, "OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 189–203.
- [28] A. Casalboni, "Aws lambda power tuning," 2024. [Online]. Available: <https://github.com/alexcasalboni/aws-lambda-power-tuning>
- [29] Kubernetes, "Configure pods and containers — kubernetes." [Online]. Available: <https://kubernetes.io/docs/tasks/configure-pod-container/resize-container-resources/> Accessed: 2025-10-07.
- [30] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX annual technical conference (USENIX ATC 18)*, 2018, pp. 133–146.
- [31] AWS Lambda, "Monitoring concurrency," 2024. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html#managing-provisioned-concurrency>
- [32] AWS, "Lambda function scaling," 2024. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/scaling.html>
- [33] Knative, "Configuring concurrency," 2024. [Online]. Available: <https://knative.dev/docs/serving/autoscaling/concurrency/>
- [34] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: rapid task provisioning with serverless-optimized containers," in *Proceedings of the 2018 USENIX Annual Technical Conference*. Boston, MA, USA, 2018, pp. 57–70.
- [35] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: a scalable and locality-enhanced framework for serverless parallel computing," in *Proceedings of the 2020 ACM Symposium on Cloud Computing*. Virtual Event, 2020, pp. 1–15.
- [36] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. Darlow, J. Wang, Q. Deng, and A. Barker, "Serverless cold starts and where to find them," in *Proceedings of the Twentieth European Conference on Computer Systems*, ser. EuroSys '25, p. 938–953.
- [37] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21, p. 64–78.
- [38] Kubernetes, "Container runtime interface (cri) — kubernetes." [Online]. Available: <https://kubernetes.io/docs/concepts/architecture/cri/> Accessed: 2025-10-07.
- [39] Kubernetes, "Open-source system for automating deployment, scaling, and management of containerized applications," 2024. [Online]. Available: <https://kubernetes.io>
- [40] MinIO, "Exascale AI Data Store," 2025. [Online]. Available: <https://www.min.io>
- [41] M. Abdi, S. Ginzburg, X. C. Lin, J. Faleiro, G. I. Chaudhry, I. Goiri, R. Bianchini, D. S. Berger, and R. Fonseca, "Palette load balancing: Locality hints for serverless functions," in *Proceedings of the Eighteenth European Conference on Computer Systems*. Rome, Italy, 2023, pp. 365–380.
- [42] S. Li, W. Wang, J. Yang, G. Chen, and D. Lu, "Golgi: Performance-aware, resource-efficient function scheduling for serverless computing," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, ser. SoCC '23, p. 32–47.
- [43] R. Gallager, *Stochastic Processes: Theory for Applications*. Cambridge University Press, 2013.
- [44] S. S. SHAPIRO and M. B. WILK, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3-4, pp. 591–611, dec 1965.
- [45] H. W. Lilliefors, "On the kolmogorov-smirnov test for normality with mean and variance unknown," *Journal of the American Statistical Association*, vol. 62, no. 318, pp. 399–402, 1967.
- [46] "CloudLab," <https://www.cloudlab.us>.
- [47] 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)*, 2022, pp. 69–84.
- [48] D. Du, Q. Liu, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Serverless computing on heterogeneous computers," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, oriLausanne, Switzerland, pp. 797–813.
- [49] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "Faas-flow: enable efficient workflow execution for function-as-a-service," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Lausanne, Switzerland, 2022, pp. 782–796.
- [50] A. Sahraei, S. Demetriou, A. Sobhghol, H. Zhang, A. Nagaraja, N. Pathak, G. Joshi, C. Souza, B. Huang, W. Cook et al., "Xfaas: Hyperscale and low cost serverless functions at meta," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 231–246.
- [51] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A scalable low-latency serverless platform," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21, p. 138–152.
- [52] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized core-granular scheduling for serverless functions," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19, p. 158–164.