ComboFunc: Joint Resource Combination and Container Placement for Serverless Function Scaling with Heterogeneous Container

Zhaojie Wen, Qiong Chen, Quanfeng Deng, Yipei Niu, Zhen Song, Fangming Liu*, *Senior Member, IEEE*

Abstract—Serverless computing provides developers with a maintenance-free approach to resource usage, but it also transfers resource management responsibility to the cloud platform. However, the fine granularity of serverless function resources can lead to performance bottlenecks and resource fragmentation on nodes when creating many function containers. This poses challenges in effectively scaling function resources and optimizing node resource allocation, hindering overall agility. To address these challenges, we have introduced ComboFunc, an innovative resource scaling system for serverless platforms. ComboFunc associates function with heterogeneous containers of varying specifications and optimizes their resource combination and placement. This approach not only selects appropriate nodes for container creation, but also leverages the new feature of Kubernetes In-place Pod Vertical Scaling to enhance resource scaling agility and efficiency. By allowing a single function to correspond to heterogeneous containers with varying resource specifications and providing the ability to modify the resource specifications of existing containers in place, ComboFunc effectively utilizes fragmented resources on nodes. This, in turn, enhances the overall resource utilization of the entire cluster and improves scaling agility. We also model the problem of combining and placing heterogeneous containers as an NP-hard problem and design a heuristic solution based on a greedy algorithm that solves it in polynomial time. We implemented a prototype of ComboFunc on the Kubernetes platform and conducted experiments using real traces on a local cluster. The results demonstrate that, compared to existing strategies, ComboFunc achieves up to 3.01 $\times$ faster function resource scaling and reduces resource costs by up to 42.6%.

Index Terms—Serverless Computing, Resource Management, Container Placement, Auto Scaling.

1 Introduction

The advent of serverless computing offers developers a fine-grained resource utilization model [1], [2]. By deploying functions on the cloud, developers can invoke them on an as-needed basis, only incurring costs for actual usage [3], [4]. This paradigm eliminates resource management overhead, enabling developers to focus on business logic, thereby significantly enhancing application development efficiency [5], [6], [7], [8], [9], [10]. Consequently, the resource management responsibility is shifted from developers to the cloud platform itself [11], [12]. Specifically, the platform needs to automatically scale function instances to meet performance demands during workload fluctuations [13], [14], [15]. Furthermore, since functions are typically encapsulated within containers, an intelligent placement mechanism is necessary to ensure efficient resource utilization [3], [13], [16], [17], [18], [19].

Achieving these goals is challenging. Firstly, function containers often experience lengthy cold starts, limiting the agility of resource scaling [20]. Secondly, serverless platforms offer various configuration options, including multi-concurrency within a single function instance, which complicates container placement. Due to the heterogeneity of function specifications, inappropriate deployment strategies can lead to significant resource fragmentation within cluster nodes, resulting in low utilization [21]. In serverless architecture, resource needs change with varying function workloads, requiring more agile and efficient resource scheduling strategies [22].

Our experiments with Knative reveal crucial insights inspiring the development of an enhanced solution. Firstly, we identify a performance bottleneck during parallel container creation on a single node, where interference between containers results in varying creation speeds across nodes. As a result, it's essential to account for node-specific container creation speeds during placement. By balancing container creation across different nodes, we can optimize parallel container creation. Existing research explores factors like data transmission affinity [23], [24], container image sharing and distribution [25], [26], and performance interference [18], [27] for container placement but often overlooks container creation speed.

Furthermore, when selecting container specifications and configuring concurrency, developers face a trade-off between using small containers with low concurrency and

- *This work was supported in part by National Key Research & Development (R&D) Plan under grant 2022YFB4501703, and in part by The Major Key Project of PCL (PCL2022A05). (Corresponding author: Fangming Liu)*
- *Z. Wen, Z. Song and Q. Deng are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab in the School of Computer Science and Technology, Huazhong University of Science and Technology, 1037 Luoyu Road, Wuhan 430074, China. E-mail: wenzhaojie@foxmail.com*
- *Q. Chen and Y. Niu are with the Central Software Institute, Distributed LAB, YuanRong Team, Huawei, China. E-mail: chenqiong13@huawei.com*
- *F. Liu is with Peng Cheng Laboratory, and Huazhong University of Science and Technology, China. E-mail: fangminghk@gmail.com*

Manuscript received xxxx xx, 2024; revised xxxx xx, 2024.

large containers with high concurrency. Our findings underscore the unique advantages of both large and small containers (Section 2.3). Consequently, we advocate for the design of a flexible resource combination strategy that leverages the strengths of both large and small containers to effectively balance resource scaling speed and utilization efficiency. This approach dynamically configures container size and concurrency based on actual conditions, enhancing overall efficiency.

Finally, we find that dynamically expanding the resource specifications of existing containers in place is more efficient than creating new ones for resource scaling [28], [29]. With the introduction of In-place Pod Vertical Scaling in Kubernetes v1.27 [30], [31], it has become feasible to modify a pod's resource without the need for pod restart. This advancement allows for increased resource allocation for existing containers and higher concurrency configurations for larger ones, facilitating swift resource scaling and enhancing resource response agility. However, if there are no active containers on the node, we still need to create new containers. Additionally, in-place pod vertical scaling depends on the available resources on the node, often limiting its benefits due to resource shortages. Therefore, careful selection of function scaling methods is crucial. Notably, existing works on vertical scaling [32], [33] do not exploit Kubernetes's in-place pod vertical scaling feature, underscoring the motivation behind our research.

Based on our insights, we introduce ComboFunc, a container resource combination and placement framework specifically designed for serverless platforms. Its key innovation lies in associating functions with heterogeneous containers of diverse resource specifications and concurrency, fully leveraging in-place pod vertical scaling. ComboFunc dynamically updates deployment plans of function containers, optimizes container placement, creates new containers as needed, identifies suitable candidates for in-place pod vertical scaling, and manages container destruction. ComboFunc not only improves function resource scaling but also reduces serverless platform costs by balancing fragmented resource utilization within the cluster.

We start by abstracting serverless function services and formally defining the container resource combination and placement problem. We introduce a function scaling cost model to evaluate various container combinations and placements. To better utilize fragmented cluster resources, we also design a function resource cost model. Given the NP-hard nature of the problem, we develop the "Cost-based Greedy Algorithm for Joint Resource Combination and Container Placement", efficiently generating optimal container resource combination and placement for each function's resource scaling in polynomial time.

To implement ComboFunc, we introduce custom resource definitions (CRDs) in Kubernetes that support heterogeneous container resource combinations and in-place vertical scaling. We also develop a prototype system of ComboFunc. Through a series of trace-based experiments on a local Kubernetes cluster, we find that ComboFunc increases the average speed of function resource scaling by up to 3.01 $\times$, reduces function resource cost by up to 42.6%, and enhances cluster resource utilization. The advantage of ComboFunc lies in its ability to not only determine the specifications and quantity of containers but also decide their placement and the method of resource scaling, whether it's creating new containers or performing vertical scaling in place.

We summarize the contributions of this paper as follows.

- We introduce ComboFunc, a strategy for jointly optimizing heterogeneous container resource combinations and placements. By balancing container creation across different nodes and integrating the capability of in-place pod vertical scaling, we enhance the agility of resource elasticity and resource utilization.
- We formally model the problem of jointly optimizing heterogeneous container resource combinations and placements for functions, and propose a heuristic algorithm based on greedy principles to solve this problem.
- We develop a prototype system based on Kubernetes and conduct experiments using trace-driven data on Knative. The results validate that our approach outperforms the most advanced strategies in terms of resource scaling speed, lower resource costs, and higher resource utilization.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation. Section 3 presents the system model and problem formulation. Section 4 describes the algorithm for heterogeneous container resource combination and placement. Section 5 describes the system's implementation. Section 6 outlines the experimental setup and presents the experimental results. Section 7 introduces related work. Section 8 discusses the limitations and future work. Section 9 provides concluding remarks.

2 Background and Motivation

2.1 Kubernetes-based Serverless Platform

Currently, mainstream open-source serverless platforms like OpenFaaS, Knative, and OpenFunction are all based on Kubernetes for managing and deploying functions. In serverless architecture, functions serve as the primary unit of execution and are usually encapsulated in containers. Each function service corresponds to a deployment and its respective service in Kubernetes. Cloud users trigger functions by sending HTTP function invocation requests to the service. During this process, the service distributes requests to multiple pod instances of the corresponding function using a load-balancing algorithm. Each function instance corresponds to a pod in Kubernetes, belonging to the respective function's deployment.

Typically, a pod can handle one function invocation request at a time. However, many serverless platforms now allow configuring maximum concurrency for functions. Function containers are designed to handle incoming requests in parallel, with each container having a specified maximum concurrency capacity. When this capacity is reached, additional containers are generated, leading to request queuing during this period, as illustrated in Figure 1.

Regarding the isolation of function resources, serverless platforms leverage Kubernetes' pod resource limits to configure and manage function resources. Kubernetes provides request and limit configurations for CPU and

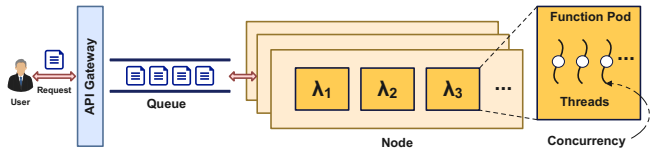


Fig. 1: Architecture of a serverless cloud platform.

memory for each pod. By setting resource limits at the pod level, serverless platforms ensure that each function instance does not consume excessive computing resources during runtime, thereby avoiding disruptions to other functions. Additionally, Kubernetes optimizes container placement based on pod resource requests. This resource-limiting mechanism effectively maintains the stability and reliability of the platform while also contributing to improved resource utilization and performance optimization.

2.2 Challenge of Serverless Resource Management

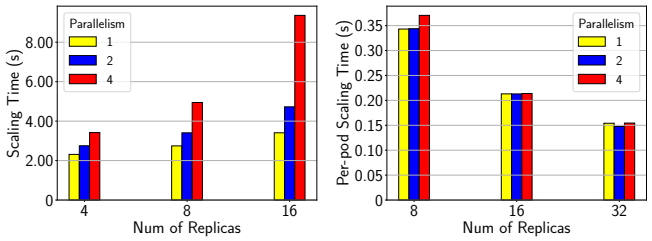
While serverless platforms are highly efficient in handling varying workloads, they also introduce specific resource management challenges. Among these challenges, ensuring that the system responds quickly and efficiently to workload changes is crucial. The agility of resource scaling and the effective utilization of cluster resources are key factors that significantly impact the performance and efficiency of serverless platforms.

Lack of scaling agility: Insufficient speed in scaling resources can contribute to a long-tail distribution of function response time. Typically, serverless platforms adopt passive autoscaling strategies to adapt to fluctuating workloads [34]. As the workload increases, the platform proactively creates additional containers to enhance concurrency capacity. Therefore, fast container resource scaling indicates that the system can quickly respond to workload changes, thereby reducing queuing delays. Requests with prolonged queuing delays contribute to the long-tail distribution of response time. Thus, improving container resource scaling agility presents a challenge for cloud platforms.

Under-utilization of node resources: The heterogeneity of containers poses a challenge in optimizing the utilization of cluster node resources. In serverless platforms, functions support various resource specifications, ranging from as little as 0.1 vCPU and 64 MB of memory to as large as 64 vCPU and 120 GB of memory [35]. Some serverless platforms even allow simultaneous configuration of CPU and memory specifications for containers, along with the maximum concurrency within a single container [36]. This diversity in container specifications increases the complexity of container placement. Therefore, it is crucial to design effective container placement strategies that can handle the heterogeneity of container configurations and enhance the resource efficiency of cluster nodes [13], [18], [21].

2.3 Preliminary Observations and Insights

Observation 1. *There is a performance bottleneck in intra-node parallel container creation, which can be improved by using inter-node parallelism to increase container creation speed. The real-time speed of container creation on different nodes varies.*



(a) Total elasticity time. (b) Per-pod elasticity time.

Fig. 2: Container scaling time under different parallelism levels.

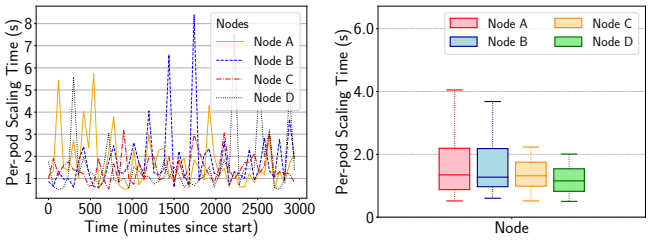
Serverless platforms employ dynamic resource scaling for functions based on their workloads, which involves the creation and destruction of multiple function containers on cluster nodes. However, non-parallelizable steps in the container creation process limit the speed of container creation on a single node [17]. To validate this observation, we conduct experiments to measure the time required to create various numbers of pods on a node within a Kubernetes cluster.

We treat creating a certain number of container replicas on a node as a scaling task. Also, we allow the node to create multiple scaling tasks and use "Parallelism" to show the number of scaling tasks running in parallel on a single node. We set the container creation numbers to 4, 8, and 16 for three scaling tasks, and set the parallelism to 1, 2, and 4. With this experiment setup, we can explore if there is performance interference between different scaling tasks on a node.

The experimental results are shown in Figure 2. From Figure 2a, we can see that when multiple parallel scaling tasks create containers on a node at the same time, their end-to-end container creation time increases proportionally. Specifically, the bars for parallelism of 1, 2, and 4 increase in height, and they increase proportionally as the number of replicas grows. This shows that creating containers for multiple deployments on the same node interferes with each other.

Additionally, we calculate the average scaling time for each container, called per-pod scaling time, by dividing the total time to create all containers by the total number of containers created. The experimental results are shown in Figure 2b. Increasing the parallelism within a node to create different numbers of containers does not significantly improve the average creation speed of each container. This shows that different function scaling tasks on the same node share the node's container creation capacity, and parallelism within a node does not effectively improve container scaling speed.

At different times, different nodes may be performing different scaling tasks, so the container creation speed on different nodes shows heterogeneity. To illustrate this, we use the Alibaba Cluster Trace [37] as the dataset for function request invocations to simulate container creation tasks on different nodes at different times in a Kubernetes cluster. Our local Kubernetes cluster consists of 5 nodes, with 1 master node and 4 worker nodes for container creation. Dur-



(a) Different timestamp. (b) Different node (all time).

Fig. 3: Container creation speed varies across time and space.

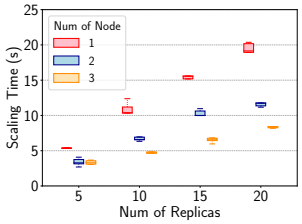


Fig. 4: Scaling time with different numbers of nodes.

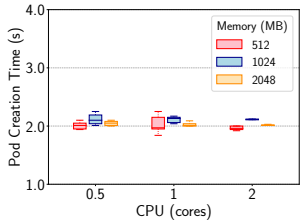
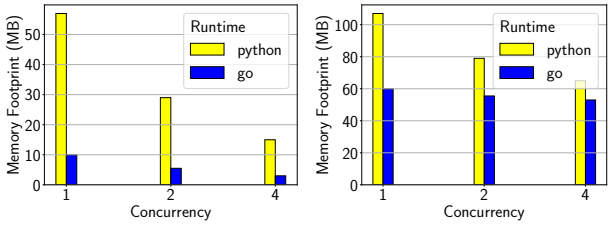


Fig. 5: Pod creation time with different container sizes.



(a) Containers in idle state. (b) Containers in busy state.

Fig. 6: Average memory footprint per concurrency under different levels of concurrency and runtime conditions.

ing the experiment, we create the corresponding number of containers in the local Kubernetes cluster according to the dataset, and we track and measure the average container creation speed on four different nodes to understand how the speed changes over time. As shown in Figure 3a, the container creation speed on different nodes shows strong volatility. The container scaling speed is affected by the elastic tasks on different nodes at different times.

However, from an average perspective, the hardware of different nodes is similar, and their container creation capabilities are comparable. Over a longer period, the status of different nodes is equivalent, so there is no significant difference in the container creation speed ability of each node in the long term. As shown in Figure 3b, although there are differences in the volatility of creation speeds on different nodes, the average creation speeds over a long period are roughly similar.

Although the container creation capacity of a single node is limited, using inter-node parallelism to create containers with multiple nodes is an effective way to improve resource scaling efficiency. To illustrate this, we set different node affinities in the deployment of Kubernetes functions to control the number of nodes participating in container creation, and measure the time required to create containers with different numbers of nodes and replicas. As shown in Figure 4, increasing the number of nodes involved in container creation helps reduce the time needed for resource scaling. This is because different nodes participate in container creation without performance interference between them. This demonstrates the necessity of timely and reasonable allocation of container creation tasks across different nodes to fully utilize the container creation capacity of different nodes.

Insight 1. *We need to select suitable nodes for container creation with inter-node parallelism, thereby improving the agility of resource scaling.*

Observation 2. *There are advantages to both large containers and small containers.*

When deploying a serverless function, we have the choice of using small containers with low concurrency or large containers with high concurrency. For instance, if a function requires 1 core and 1 GB of memory for a single concurrent execution, we can either select the minimum container size of 1 core and 1 GB of memory configured with concurrency of 1, or we can opt for larger containers with 4 cores and 4 GB of memory, configured with concurrency of 4. While the latency of requests may slightly differ between containers of different sizes, they can provide the same concurrent processing capacity. Figure 5 demonstrates that there is no significant disparity in container creation time when using different container sizes on a node. This implies that configuring larger containers enables faster resource scaling when creating the same number of containers.

Moreover, larger containers exhibit lower average resource usage per individual function concurrency. We compare the average memory footprint per single concurrency for containers of different sizes in both idle and busy states. As depicted in Figure 6a, it can be observed that in the idle state, the average memory footprint per concurrency decreases proportionally as the container size increases. This is because containers of different sizes have similar resource footprints when idle, and larger containers can distribute the resource overhead across all concurrent executions. Additionally, Figure 6b shows that even in the busy state, when containers reach their maximum concurrent executions, the average resource footprint per single concurrency remains lower for larger containers. This indicates that larger containers offer a better performance-to-resource ratio compared to smaller containers, making them more cost-effective in terms of resource utilization.

However, larger containers also have drawbacks. Firstly, using larger containers can result in increased resource fragmentation within the cluster, as illustrated in Figure 7, where placing containers of different sizes on a node leads to varying levels of resource fragmentation. Secondly, the scheduling criteria for larger containers are more stringent, making them less flexible compared to smaller containers. As a result, larger containers have fewer placement options, which can impact inter-node parallelism during container

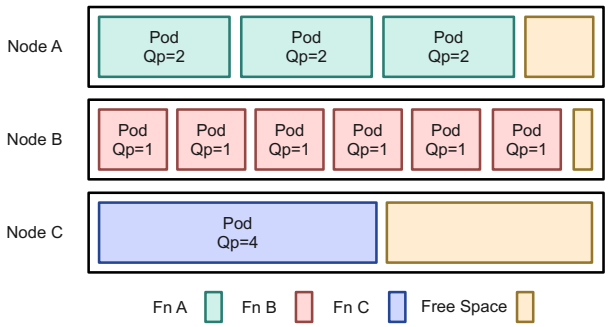


Fig. 7: Placing containers of different sizes on different nodes. A single container size can lead to resource fragmentation, while smaller containers can better utilize node resource fragments.

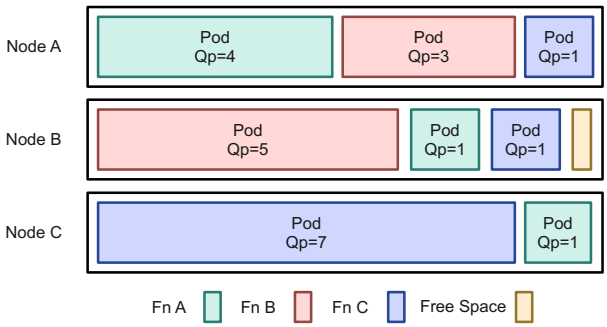


Fig. 8: Combining resources for heterogeneous function containers. Mapping functions to containers of various sizes can effectively leverage node resource fragments and improve resource utilization.

creation and hinder resource elasticity speed.

Insight 2. *It motivates us to utilize heterogeneous containers, combining the benefits of both large and small containers by enabling a single function to correspond to containers of varying sizes.*

As demonstrated in Figure 8, each colored square represents a different function pod, and requests for the same function can be routed to containers with different resource specifications and concurrency. Compared to the homogeneous container combination setup depicted in Figure 7, the heterogeneous container combination within the same three nodes can increase total function concurrency and improve cluster resource utilization.

Observation 3. *Compared to creating new containers for resource scaling, dynamically adjusting the resource specifications of existing containers is a more efficient approach.*

In v1.27, Kubernetes introduced a new alpha feature called "In-place Pod Vertical Scaling", which allows users to adjust the CPU and memory resources allocated to pods without restarting the containers. This improvement significantly enhances the flexibility and efficiency of dynamic resource management in Kubernetes. The API now permits modifications to the CPU and memory resources under the resources field of pod containers. This means users can adjust resources by patching the specifications of running pods without needing to recreate or restart them. This new feature not only reduces the complexity of resource adjustments but also avoids unnecessary service disruptions. Additionally, recent work [29] has also achieved dynamic allocation and adjustment of container resources at the kernel level. Consequently, we can increase the resource allocation of existing containers and adjust their concurrency configurations without restarting them, enabling quick resource scaling and improving the agility and flexibility of resource scaling.

Nevertheless, this approach is not without limitations. In-place pod vertical scaling necessitates the presence of warm containers. In the absence of warm containers, new containers must still be created. Additionally, the maximum size for in-place container vertical scaling is contingent upon the available resources within the node. In many instances, nodes may lack adequate resource space, thereby diminishing the effectiveness of this approach.

Insight 3. *To achieve efficient resource elasticity, a combination of both container creation and in-place pod vertical scaling is required.*

Building upon these observations and insights, we introduce the core concept of ComboFunc: configuring a single function to utilize containers with diverse resource specifications and concurrency settings. This strategy fully leverages the potential for parallel container creation and in-place vertical scaling on different nodes. The objective is to enhance resource scaling agility by effectively harnessing the advantages offered by both large and small containers, while optimizing the utilization of cluster resources. Through the joint optimization of container resource combinations and placements, an optimal resource scheduling scheme can be achieved.

3 System Model and Problem Formulation

In this section, we define the container resource combination and placement model to further explore the research problem. The main notations are shown in Table 1.

3.1 Deployment Plan of Function Containers

In our research, we primarily focus on two key resource dimensions: CPU and Memory. These resources are central to container scheduling and are widely supported by mainstream container resource schedulers [38], [39], [40]. Although other resource dimensions, such as micro-architecture-level cache, memory bandwidth, and disk I/O, are also important for scheduling optimization, we decided to exclude these aspects from our study to simplify its scope.

We consider a cluster consisting of many server nodes. Let $\mathcal{N}$ be the set of nodes, where n represents a specific node, and $n \in \mathcal{N}$. Each node in our cluster has a CPU capacity of C_n and a memory capacity of M_n. Each node may host many services or functions, occupying certain resources. Therefore, we define C_n^r and M_n^r as the remaining CPU and memory capacities of node n, respectively. Against this backdrop, we focus on the resource scaling of a serverless function f, aiming to create containers for a function f on different nodes to meet its target concurrency. As function f correlates with containers of various sizes, we denote the set of all available container specifications for function f as

TABLE 1: Main Notations

Notation	Definition
$\mathcal{N}$	The set of serverless cloud platform cluster nodes
n	A server node
C_n	CPU capacity of node n
M_n	Memory capacity of node n
C_n^r	CPU remaining capacity of node n
M_n^r	Memory remaining capacity of node n
f	A serverless cloud function
P_f	Set of container specifications for f
C_p	CPU specification of container p
M_p	Memory specification of container p
Q_p	Concurrency specification of container p
Q	Total concurrency before scaling
Q'	Total concurrency after scaling
X_{np}	Number of containers with specification p on node n
$R_f(Q)$	Deployment plan of function f at concurrency of Q
S_{np}	Scaling cost of deploying pod p on node n
B_{np}	Resource cost of deploying pod p on node n
D_{np}	Deployment cost of deploying pod p on node n
S_f	Total scaling cost of function f
B_f	Total resource cost of function f
D_f	Total deployment cost of function f
I_p	Image pulling overhead of container p
λ	Trade-off parameter between scaling and resource cost

P_f. For a specific container specification $p \in P$, we use C_p to represent its CPU size, M_p to represent its memory size, and Q_p to represent its configured concurrency. We use X_{np} to represent the number of containers with specification p deployed on node n, then we have:

$$\begin{cases} X_{np} = 0, & \text{if container } p \text{ is not deployed on node } n \\ X_{np} \geq 1, & \text{if container } p \text{ is deployed on node } n \end{cases} \tag{1}$$

The sum of the CPU and memory size of function containers on each node n must not exceed the remaining CPU and memory capacities of node n. Hence, we have the following constraints:

$$\forall n \in \mathcal{N}, \sum_{p \in P_f} X_{np} C_p \leq C_n^r \tag{2}$$

$$\forall n \in \mathcal{N}, \sum_{p \in P_f} X_{np} M_p \leq M_n^r \tag{3}$$

We define the deployment plan of function f as $R_f = \{(n, p, X_{np}) | n \in N, p \in P_f\}$, which records the container deployment of function f in the cluster. It includes the placement of containers with different specifications.

Ideally, there should be a container available to handle each function request upon arrival. To achieve this goal, we can calculate the required target concurrency based on the actual function workload.

Therefore, given the target total concurrency for a function, the key is to find a reasonable container deployment plan. Let Q represent the target total concurrency of all function containers. Then we have:

$$\sum_{p \in P_f} Q_p \geq Q \tag{4}$$

Definition 1. *We define the state $R_f(Q)$ as a feasible container deployment plan for function f with a total concurrency of Q if and only if R_f satisfies (1)(2)(3)(4).*

According to Definition 1, we understand what container deployment plan can meet the function concurrency target. When the workload changes, the target concurrency of the function also changes accordingly. Suppose the current container deployment state is $R_f(Q)$, and the scaling task is to improve function concurrency from Q to Q'. In that case, we need to find a new feasible container deployment plan $R_f(Q')$. In other words, for different levels of target concurrency, we need to find a corresponding deployment plan that matches it. The cloud platform's task is to update the existing container deployment plan by creating, destroying, and in-place scaling containers to align with the target concurrency.

Finding a feasible deployment plan based on a given target concurrency is not difficult. The real challenge lies in finding a more optimized deployment plan. Next, we define what constitutes a good deployment plan.

3.2 Scaling Cost Model

As discussed earlier, when cloud platforms perform resource scheduling for serverless functions, they focus on improving the agility of resource scaling. The process of resource scaling is essentially the transition from target concurrency Q to Q'. Therefore, we establish a resource scaling cost model to evaluate the effectiveness of transitioning from the deployment plan $R_f(Q)$ to $R_f(Q')$.

To describe the agility of resource scaling, we need to find an indicator that characterizes the speed of function concurrency changes. We know that the rate of increase in function concurrency directly affects the distribution of function delays. Therefore, we define the scaling cost S_f as the average time it takes for the function f to complete the update of its target concurrency, denoted as S_f. For a container in spec p to be scheduled on node n, its scaling cost is denoted as S_{np}, and we have:

$$S_f = \frac{1}{\sum_{p \in P_f} Q_p} \sum_{n \in \mathcal{N}} \sum_{p \in P_f} X_{np} \cdot S_{np} \tag{5}$$

We know that if a container instance of function f exists on a node, we can choose to perform in-place pod vertical scaling, or simply create new containers. In the case of in-place pod vertical scaling, we introduce the time cost on node n as τ_n^u. For the case of creating new containers, we introduce the time cost of creating new containers on node n as τ_n^c. We know that during the process of container creation, there is a time expense of pulling the container image through the network from the image repository, which we denote as I_p. However, for in-place pod vertical scaling, the container image does not need to be downloaded. Therefore, we can express S_{np} under different conditions as follows:

$$S_{np} = \begin{cases} \tau_n^u, & \text{if } p \text{ is in-place vertical scaling} \\ \tau_n^c \cdot k_n + I_p, & \text{if } p \text{ is created} \end{cases} \tag{6}$$

where k_n represents the number of inter-node creating containers to show resource contention during the container creation process. Now, we find an indicator to evaluate the agility of function resource scaling. Smaller S_f indicates lower resource scaling time and faster resource scaling speed.

3.3 Resource Cost Model

As mentioned earlier, when cloud platforms make resource scheduling decisions, their primary consideration is to reduce the resource costs [41], [42]. Therefore, we need to find a method to evaluate the resource cost. Due to the presence of different sizes of resource fragments in serverless cluster nodes, their utilization difficulty varies. Consequently, the resource cost for smaller resource fragments is relatively cheaper compared to larger ones. It is similar to the production of silicon wafers for chips, where the unit price increases for selecting larger area silicon wafers [43] because of its scarcity. Based on this principle, we define a function resource cost model. We link the prices of different node resources to their resource utilization rates. When a node's resource utilization rate is high, we apply an additional discount to the price, and when the resource utilization rate is low, we introduce a price hike. As a result, we propose the following:

$$\beta_n^c = \frac{C_n^r}{C_n} \cdot (1+x) + (1 - \frac{C_n^r}{C_n}) \cdot (1-x) \tag{7}$$

$$\beta_n^m = \frac{M_n^r}{M_n} \cdot (1+x) + (1 - \frac{M_n^r}{M_n}) \cdot (1-x) \tag{8}$$

Where $x \in [0,1]$ is the maximum percentage of price fluctuation, β_n^m represents the memory price of node n, and β_n^c represents the CPU price of node n. When the resource utilization is 0%, the price reaches the highest. When the utilization is 100%, the price reaches the lowest.

We denote the resource cost of a container with specification p on node n as B_{np}, then we have:

$$B_{np} = \beta_n^c \cdot C_p + \beta_n^m \cdot M_p \tag{9}$$

For function f, its total resource cost is B_f, which is the sum of the resource costs of all its containers, given as:

$$B_f = \frac{1}{\sum_{p \in P_f} Q_p} \sum_{n \in \mathcal{N}} \sum_{p \in P_f} B_{np} \tag{10}$$

With the definition of the function resource cost, we can guide the resource scheduler to prioritize fragmented resources in the cluster by minimizing the resource cost, achieving efficient utilization, and cost optimization of cluster resources.

3.4 Problem Definition

In the previous sections, we establish two models to describe the resource scaling agility and the resource cost of the functions. Our problem essentially seeks a new container deployment plan while giving the current function container plan to minimize both function scaling cost and function resource cost. Since there are two objectives to minimize, we define the deployment cost D_f as a unified optimization objective, which is the weighted sum of the two costs:

$$D_f = S_f + \lambda \cdot B_f \tag{11}$$

where λ is the trade-off parameter used to adjust the importance between scaling and resource cost. Now, we can define the optimization problem to minimize the deployment cost. Thus, our **F**unction **R**esource **C**ombination and **P**lacement Problem (**FRCP**) is defined as follows:

Definition 2. *Problem **FRCP** is that given the current deployment plan $R_f(Q)$ of function f, find a new deployment plan $R_f(Q')$ with the corresponding total concurrency Q':*

$$\min D_f \tag{12}$$
$$\textit{subject to: } (1)(2)(3)(4)(5)(6)(7)(8)(9)(10)(11)$$

In this problem, we aim to find a new container deployment plan that meets the given target concurrency Q' while minimizing the deployment cost, which is a weighted combination of scaling cost and resource cost. The trade-off parameter λ allows for flexibility in determining the trade-off between these two optimization objectives.

3.5 Problem Complexity

We know that even for a simple resource scheduling problem, finding the optimal solution is NP-hard. The **FRCP** studied in this paper is no exception. We provide Theorem 1 to prove the computational complexity of the problem.

Theorem 1. *Problem **FRCP** is NP-hard.*

Proof. The multi-dimensional multiple knapsack problem is a classic optimization problem known to be NP-hard. Its definition is as follows: Given a set of items N, each item $i \in N$ has multiple attributes, such as weight w_i and volume e_i. The set of knapsacks is M, and each knapsack $j \in M$ has multiple capacity constraints, such as weight capacity W_j and volume capacity E_j. We define the value of each item i as v_i. We define a non-negative integer matrix x, where x_{ij} represents the quantity of item i placed in knapsack j. The objective is to maximize the total value while satisfying multiple-dimensional constraints:

$$\max \sum_{i \in N} \sum_{j \in M} v_i x_{ij} \tag{13}$$

$$\text{subject to : } \begin{cases} \sum_{i \in N} w_i x_{ij} \leq W_j, \forall j \\ \sum_{i \in N} e_i x_{ij} \leq E_j, \forall j \\ x_{ij} \in \mathbb{Z}^+ \end{cases} \tag{14}$$

We consider simplifying our problem in Definition (2) by removing the (4) constraint. Given an instance $A = (N, M, v_i, x_{ij}, w_i, e_i, W_j, E_j)$ of the knapsack problem, we can map it to a simplified instance $A' = (P, N, -D_{np}, x_{np}, C_p, M_p, C_n, M_n)$ of our problem. The above mapping can be done in polynomial time. Therefore, we have constructed a polynomial-time reduction from the multidimensional knapsack problem to our problem. If there exists an algorithm that solves problem A', then it can also solve the corresponding knapsack problem. Thus, the multidimensional knapsack problem can be considered a special case of our problem. Given the NP-hardness of the multidimensional knapsack problem, Problem **FRCP** is also NP-hard. □

4 Solution

In the previous sections, we analyze the complexity of the problem. Due to the large scale of nodes and function containers in the cluster, finding the global optimal solution can be a very time-consuming task. To overcome this challenge,

we adopt a cost-based greedy heuristic algorithm to reduce computational complexity.

4.1 Cost-based Greedy Algorithm

Our goal is to find a deployment plan for function containers under resource constraints to minimize the total deployment cost while ensuring the total concurrent capacity of the containers reaches the target value. We realize that the primary objective of the problem is to minimize deployment costs. Therefore, we select the container specifications and node combinations with the smallest deployment cost per unit concurrency for deployment. We denote the deployment cost of container specification p on node n as D_{np}, $D_{np} = S_{np} + \lambda \cdot B_{np}$. Thus, D_{np}/Q_p represents the deployment cost per unit concurrency. The smaller this value, the higher the cost-effectiveness of the container.

By gradually selecting the most cost-effective containers until the total concurrency of the functions reaches the target value, we ultimately obtain an optimized container deployment plan. Despite the intuitive nature of this approach, the advantages of the greedy algorithm lie in its simplicity and efficiency. By quickly selecting container deployment combinations with minimal costs, we can obtain an approximate optimal solution within a reasonable time.

Next, we provide a detailed explanation of the steps of the cost-based greedy algorithm, as shown in Algorithm 1:

Algorithm 1: Cost-based Greedy Algorithm for Joint Resource Combination and Container Placement

Require: Deployment status $R_f(Q)$, elastic target concurrency Q'
Ensure: Deployment status $R_f(Q')$
1: Initialize $R_f(Q') = \{(n, p, X_{np}) | X_{np} = 0, n \in N, p \in P_f\}$
2: Initialize current total concurrency $Q_{curr} = 0$
3: **while** $Q_{curr} \leq Q'$ **do**
4: Current minimum value of $D_{np}/Q_p \rightarrow D_{min}$
5: Currently selected node N_{curr} and pod spec P_{curr}
6: **for** $(n, p, X_{np}) \in R_f(Q')$ **do**
7: **if** $C_p \geq C_n^r$ or $M_p \geq M_n^r$ **then**
8: **continue**
9: **end if**
10: **if** $D_{np}/Q_p < D_{min}$ **then**
11: $D_{min} \leftarrow D_{np}/Q_p$
12: $P_{curr} \leftarrow p$
13: $N_{curr} \leftarrow n$
14: **end if**
15: **end for**
16: Update $R_f(Q')$ with $(N_{curr}, P_{curr}, X_{np}) \leftarrow (N_{curr}, P_{curr}, X_{np} + 1)$
17: $C_n \leftarrow C_n - C_p$
18: $M_n \leftarrow M_n - M_p$
19: $Q_{curr} \leftarrow Q_{curr} + Q_p$
20: **end while**
21: **return** $R_f(Q')$.

In Algorithm 1, we have a set of container configurations denoted by P_f, and a set of nodes denoted by $\mathcal{N}$. Each container p has a deployment status X_{np} on node n, a deployment cost D_{np}, a concurrency rate Q_p, and CPU and memory requirements C_p and M_p respectively.

In Line 1, we initialize the return value $R_f(Q')$, where all X_{np} values are set to 0, indicating that no containers are deployed initially. In Line 2, we initialize the concurrency as zero, which means no container is selected. Starting from Line 3, a while loop begins, and in Line 4, D_{min} is created to store the current best cost-effectiveness ratio. Lines 6 to 15 iterate through all combinations of container specs and nodes, updating the best cost-effectiveness ratio for the selected container spec and node. Line 16 is responsible for updating the container deployment plan in $R_f(Q')$, and Lines 17 to 19 are used to update the node status. When the target concurrency is exceeded, Line 21 returns the optimized result of the container deployment plan.

4.2 Algorithm Complexity

Theorem 2. *The time complexity of Algorithm 1 is $O(Q \cdot |N| \cdot |P_f|)$, where Q represents the target concurrency of function, $|N|$ represents the number of nodes, and $|P_f|$ represents the number of available container specifications of function.*

Proof. In Algorithm 1, the main loop is a while loop with the termination condition $Q_{curr} \leq Q'$. In each iteration of the loop, we iterate through all combinations of elements in the node-set N and the container configurations P_f, performing a series of operations with time complexity $O(1)$. Since the minimum concurrency of containers is 1, in the worst case, the loop will execute Q times. In the worst case, the total number of possible container deployment combinations is $|N| \times |P_f|$. Therefore, the time complexity of the algorithm is $O(Q \cdot |N| \cdot |P_f|)$. □

Theorem 2 demonstrates the time complexity of our algorithm. It indicates that our solution can solve the problem in polynomial time.

5 Implementation

In this section, we delve into the implementation details of the entire system, known as ComboFunc.

5.1 System Overview

ComboFunc is a resource scaling system developed using Go and Python, and it is built on top of Knative. The code base comprises approximately 3.2K lines of code. The system architecture of ComboFunc is depicted in Figure 9. The blue components represent the elements specific to ComboFunc, while the uncolored components are inherited from Knative Serving, which serves as the underlying Function-as-a-Service (FaaS) platform. The central decision-making component of ComboFunc is the Resource Scaler, responsible for executing Algorithm 1. The Metrics Collector collects invocation data for various functions from the API Gateway, calculates the Target Function Concurrency for each function, and communicates this information to the Resource Scaler for informed scaling decisions tailored to heterogeneous containers.

The Load Balancer ensures equitable distribution of traffic across the heterogeneous containers, thus enabling effective load balancing. Conversely, the Container Scheduler acts as the container scheduler within the Kubernetes cluster. It executes our deployment strategy, optimizing the placement of containers on nodes within the cluster. Furthermore, it leverages the in-place pod vertical scaling feature to efficiently adjust resource scaling for containers, resulting in more agile resource management.

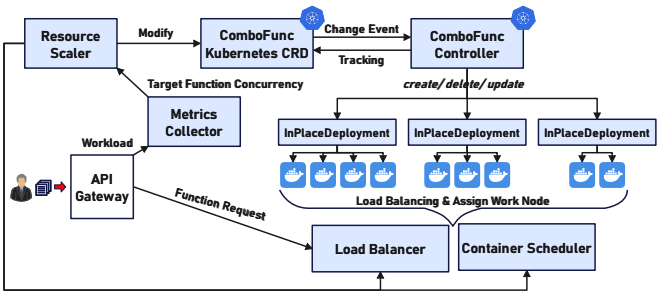


Fig. 9: System architecture of ComboFunc.

5.2 ComboFunc CRDs and Controller

Function scaling involves instructions for retaining, deleting, or in-place vertical scaling of existing containers, as well as creating any necessary new containers. The implementation of a function scaling involves sequentially updating the state of all containers associated with a function based on pod specifications. To simplify the management of containers belonging to the same function by the Resource Scaler, we utilize Kubernetes Custom Resource Definition (CRD) to define ComboFunc resources. A ComboFunc object represents a function that encompasses multiple pods with different specifications. In traditional Kubernetes, a group of containers with identical specifications is typically defined using deployments. Consequently, a ComboFunc object is a collection of several deployments. An example YAML file of a ComboFunc resource object's CRD is provided in Figure 10a, illustrating a ComboFunc with three deployments that vary in CPU, Memory, and Concurrency specifications, corresponding to distinct values under the $functions$ field in the diagram.

The ComboFunc Controller employs Kubernetes' list-watch mechanism to continuously monitor updates to ComboFunc resources, enabling real-time creation, deletion, or modification of deployment resource objects. However, a challenge arises because Kubernetes' in-place pod vertical scaling mechanism is designed for pod resource objects and is not directly supported by deployments. To overcome this limitation, we have introduced InPlaceDeployments through CRD and developed a CRD Controller to enable pods within deployments to support the in-place vertical scaling feature.

InPlaceDeployments follow a similar format to native deployments, as demonstrated in Figure 10b. The InPlaceDeployment Controller's logic is implemented using the reconciliation mechanism, akin to native deployments. The key difference lies in the update process for pods. Instead of performing rolling updates by deleting and creating new pods, the InPlaceDeployment Controller facilitates in-place upgrades of pods, preserving existing pods whenever possible. It is important to note that all mentioned CRD Controllers have been developed using Kubebuilder, the official scaffolding framework provided by Kubernetes, ensuring consistency and compatibility within the Kubernetes ecosystem, as depicted in Figure 11.

```yaml
apiVersion: xxx.combofunc.com/v1
kind: ComboFunc
metadata:
  name: test-combofunc
  labels:
    app: test-combofunc
  namespace: default
spec:
  functions:
    - resources:
        cpu: "100m"
        memory: "100Mi"
      concurrency: 1
      replicas: 2
      image: xxx
    - resources:
        cpu: "200m"
        memory: "200Mi"
      concurrency: 2
      replicas: 1
      image: xxx
    - resources:
        cpu: "100m"
        memory: "300Mi"
      concurrency: 3
      replicas: 3
      image: xxx
```

(a) ComboFunc CRD.

```yaml
apiVersion: xxx.combofunc.com/v1
kind: InPlaceDeployment
metadata:
  name: test-inplacedeployment
  labels:
    app: test-inplacedeployment
  namespace: default
spec:
  replicas: 4
  template:
    spec:
      containers:
      - name: video-processing
        image: video-processing:latest
        resizePolicy:
        - resourceName: "cpu"
          restartPolicy: "NotRequired"
        resources:
          limits:
            cpu: "1000m"
            memory: "1000Mi"
          requests:
            cpu: "1000m"
            memory: "1000Mi"
```

(b) InPlaceDeployment CRD.

Fig. 10: Custom Resource Definition (CRD) for InPlaceDeployment and ComboFunc.

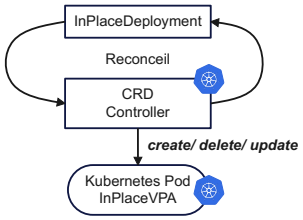


Fig. 11: Workflow diagram of the InPlaceDeployment Controller.

5.3 Container Scheduler and Load Balancer

In the decision-making process of ComboFunc, the placement of each container is not only considered but also accomplished through a highly sophisticated Container Scheduler. This scheduler has been meticulously designed to accurately determine the optimal position of each container within the cluster, thereby ensuring optimal resource utilization. To achieve this objective, our unique scheduling algorithm has been seamlessly integrated into the Kubernetes Scheduler. During the actual scheduling process, we strictly adhere to the container placement plan provided by the Resource Scaler, guaranteeing that each container is positioned in its most suitable location.

During the system's operation, user requests are incessantly sent to the API Gateway of the FaaS platform. This API Gateway serves as the central entry point of the system, responsible for receiving and processing all incoming requests. Upon receiving a request, the API Gateway promptly forwards it to the Load Balancer. The Load Balancer, being more than just a simple traffic distributor, intelligently assigns requests to the respective pods based on specific traffic weights. This implies that the volume of requests received by each pod dynamically adjusts according to their processing capabilities and current workload.

Furthermore, each pod is configured with strict container concurrency limits. These concurrency limits ensure that each container can handle only a predefined number of requests at any given time. This resource management strategy effectively prevents any individual container from experiencing performance degradation due to overload. The

Load Balancer also takes on the responsibility of real-time monitoring of the current concurrency level of each container. Once a container reaches its concurrency limit, the Load Balancer promptly halts the allocation of new requests to that container. This intelligent scheduling mechanism ensures the efficient operation of the entire system, preventing resource waste and system overload.

6 Experimental Evaluation

In this section, we demonstrate the advantages of ComboFunc through a series of experiments. We aim to answer the following research questions (RQs):

- **RQ1:** Can ComboFunc offer better resource scaling agility compared to other strategies?
- **RQ2:** Can ComboFunc reduce function resource cost compared to other strategies?
- **RQ3:** Can ComboFunc improve cluster resource utilization compared to other strategies?
- **RQ4:** How does the adjustment of the trade-off parameter in ComboFunc influence the balance between resource scaling agility and resource cost?
- **RQ5:** How does the number of nodes in a cluster affect ComboFunc's performance in terms of function resource scaling speed and resource cost?
- **RQ6:** What is the impact of individual container creation speed on the resource scaling speed and resource cost of ComboFunc and other strategies?
- **RQ7:** What is the decision-making cost of the ComboFunc algorithm?

6.1 Experimental Setup

TABLE 2: Testbed configurations.

Name	Configuration
CPU	Dual AMD EPYC 7R12 Processor with 96 cores 192 threads (@2.5GHz)
Memory	256GB DDR4 2133 MHz with 16 channels
Disk	Intel P4501 4TB
Network	External network bandwidth over 1Gbps
OS	Ubuntu 20.04 LTS with kernel 5.15.0-94-generic
VM	QEMU emulator version 4.2.1
Kubernetes	v1.27.1 with feature gate InPlacePodVerticalScaling
Containerd	v1.6.18
Knative	knative-serving v1.7

In this section, we introduce the detailed setup of the ComboFunc experiments.

- **Testbed:** To evaluate our heterogeneous container resource combination and scheduling framework, we prepare a real Kubernetes cluster consisting of 16 nodes, each configured with 8 CPU cores and 16GB of memory. The device parameters of the experimental platform are shown in Table 2.
- **Worker functions:** To simulate a real FaaS production environment, we deploy a series of functions on the cluster and generate diverse workloads to invoke these functions. During the experiment, due to the influence of different function workloads, these functions need to perform function scaling tasks on different nodes. In terms of functions for evaluation, we choose four different types of functions from [44], [45]: File Processing (FP), Video Processing (VID), Web Service (WEB), and Machine Learning (ML). Each function has unique resource requirements and can be configured with different levels of concurrency.
- **Workload traces:** We select workload traces from the Azure Function Invocation Dataset [46], which exhibits certain bursty workload changes, to test the performance of ComboFunc in terms of resource scaling. We implement a multi-threaded workload generator tailored for the FaaS platform to simulate the real workload of these four functions. In actual experiments, to ensure that each function has sufficient resources under different workloads, we standardize and scale the function workload in advance to match the available resources of the local testbed, and any additional requests exceeding the cluster's resource capacity are discarded according to the principle of fairness.

During the experiment, we observe and record the resource scaling time and cost of the four functions at various timestamps. We select the following baselines for comparison with ComboFunc:

- **Knative [47]:** The default scheduling strategy of the serverless platform Knative. It is based on a kube-scheduler for scheduling and utilizes homogeneous containers without in-place pod vertical scaling.
- **Autopilot [13]:** Google's internal cluster scheduling method for the Borg [40]. It supports in-place vertical scaling of container resources but does not utilize heterogeneous containers.
- **ComboFunc w/o In-place:** The strategy of ComboFunc utilizes heterogeneous containers without in-place pod vertical scaling.
- **Random Fit [48]:** A scheduling algorithm that determines the placement of containers by selecting the first available node that can accommodate the resource requirements of the container, without in-place pod vertical scaling and heterogeneous containers.
- **Trimaran [49]:** An advanced scheduling strategy enabling the Kubernetes scheduler to minimize machine costs by understanding the gap between resource allocation and actual resource utilization. It balances the actual average workload and mitigates the risk associated with sudden workload fluctuations.

The comparison between different methods can be seen in Table 3.

6.2 Improvement of Scaling Agility (RQ1)

We evaluate the performance improvement of ComboFunc in terms of function resource scaling speed. As shown in Figure 12, we analyze the average scaling time for the four function containers deployed in the cluster. Compared to other strategies, ComboFunc demonstrated significant advantages, exhibiting the lowest average scaling time. Specifically, compared to Random Fit, ComboFunc speeds up the

TABLE 3: Comparison of methods.

Method	In-place VPA	Heterogeneous Spec	Scheduling Algorithm
Knative	No	No	Default kube-scheduler
Autopilot	Yes	No	Default kube-scheduler
ComboFunc w/o In-place	No	Yes	ComboFunc
ComboFunc	Yes	Yes	ComboFunc
Random Fit	No	No	Random select a node that fits
Trimaran	No	No	LoadVariationRiskBalancing

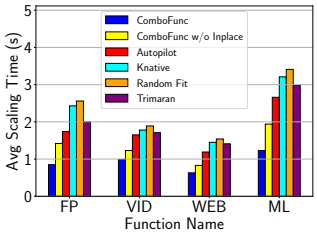


Fig. 12: Average function scaling time with different functions.

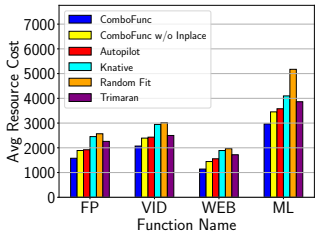


Fig. 13: Average function resource cost with different functions.

four functions by 3.01×, 1.91×, 2.44×, and 2.77×, respectively. Compared to Knative, the speedups are 2.85×, 1.80×, 2.30×, and 2.61×, respectively. Compared to Autopilot, the speedups are 2.05×, 1.67×, 1.89×, and 2.16×, respectively. Compared to Trimaran, the speedups are 2.35×, 1.73×, 2.24×, and 2.42×, respectively.

Among all these strategies, Random Fit exhibits the lowest function resource scaling speed. This is because it only considers whether available resources on different nodes meet the requirements, without taking into account the differences in container creation speed on different nodes. Therefore, randomly selected nodes may suffer from performance interference during the container creation process, leading to performance bottlenecks. This further underscores the motivation behind ComboFunc, which is to make targeted scheduling optimizations based on the actual capacity of container creation on different nodes. In addition, ComboFunc significantly improves the resource elasticity scaling speed compared to strategies that do not enable in-place pod vertical scaling, increasing by 40.1%, 19.5%, 24.0%, and 36.6% for these four functions, respectively. This indicates that leveraging the latest features of Kubernetes, which allow dynamic adjustments of container resource allocation without the need for container recreation, is a crucial means to enhance resource elasticity and agility.

Figure 14 shows the changes over time in function concurrency, number of replicas, and scaling time, the key to understanding how ComboFunc adapts dynamically to drastic workload fluctuations. It shows significant fluctuations in both concurrency and the number of replicas, with the ratio of replicas to concurrency varying between 1:3 and 1:6. This fluctuation demonstrates ComboFunc's flexibility in resource configuration through various configurations of container concurrency and combinations of differently sized heterogeneous containers. Moreover, the trend in scaling time is smoother compared to concurrency and replicas, indicating that ComboFunc's strategy of optimizing parallel

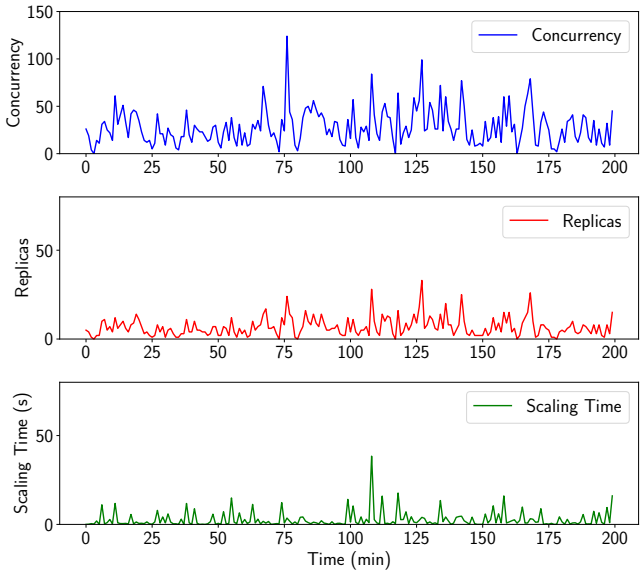


Fig. 14: Temporal dynamics of function concurrency, replicas, and scaling time.

container creation and in-place pod vertical scaling effectively reduces bursty spikes in scaling time, thereby enhancing system stability and response speed. However, there are moments when scaling time spikes, likely due to cluster resource limits not fully utilizing ComboFunc's advantages, leading to bottlenecks in container creation. This shows that ComboFunc's strategy is not infallible. When faced with limits, ComboFunc's ability may be challenged.

6.3 Reduction of Resource Cost (RQ2)

We evaluate the resource cost reduction of ComboFunc, encompassing both CPU and memory costs. As shown in Figure 13, ComboFunc shows the lowest average resource cost among different functions, indicating its advantage in cost optimization. Specifically, compared to Random Fit, ComboFunc reduces the average resource overhead for these four functions by 38.5%, 31.2%, 41.9%, and 42.6%, respectively. Compared to Knative, the reductions are 35.6%, 29.7%, 39.6%, and 27.5%, respectively. Compared to Autopilot, the reductions are 17.8%, 14.8%, 26.7%, and 17.1%, respectively.

Additionally, we observe that the extent of resource savings varies among different functions. This is because different functions have distinct performance requirements and characteristics, leading to varying demands for resource scaling speed and resource cost. We find that ComboFunc's effectiveness is more pronounced when dealing

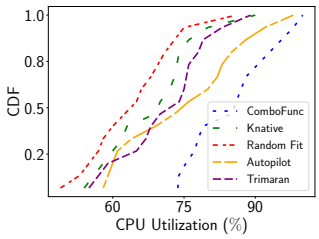


Fig. 15: CDF of node CPU utilization ratio with different strategies.

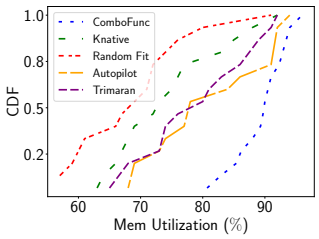


Fig. 16: CDF of node memory utilization ratio with different strategies.

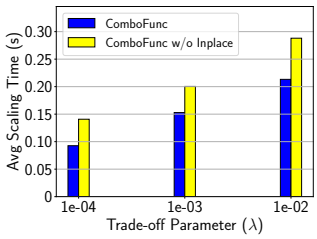


Fig. 17: Average function scaling time with different trade-off parameters.

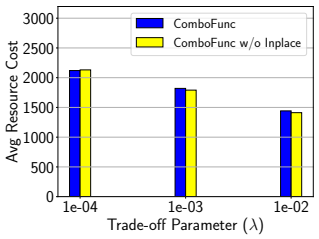


Fig. 18: Average function resource cost with different trade-off parameters.

with resource-intensive function containers. This is because improving scaling speed is a pain point for such function applications, and our heterogeneous resource combination mechanism effectively enhances scaling agility while reducing resource fragmentation in these cases.

6.4 Improvement of Resource Utilization (RQ3)

We assess the improvement in cluster node resource utilization achieved by ComboFunc. At different times, as different functions undergo resource scaling, the resource utilization of various nodes in the cluster also changes. We compile probability distribution CDF graphs for CPU and memory utilization of different nodes at different timestamps, as shown in Figure 15 and Figure 16. The closer the quantile curve in the graph is to the lower right corner, the higher the proportion of nodes with high resource utilization. It can be observed that, compared to other strategies, ComboFunc exhibits higher overall resource utilization. This is not only because ComboFunc optimizes scheduling by leveraging the combination of heterogeneous container resources to better utilize resource fragments within the cluster but also because ComboFunc's resource scaling is more agile, thereby enhancing the efficiency of resource utilization across the cluster.

6.5 Impact of Different Trade-off Parameters (RQ4)

Due to the inherent trade-off between high resource scaling agility and low resource costs, we incorporate a tradeoff parameter into our model. To investigate the impact of selecting different trade-off parameters, we conduct experiments by varying the trade-off parameter. The results are illustrated in Figure 17 and Figure 18. Choosing different trade-off parameters implies different preferences for scaling agility and resource cost. A larger parameter value corresponds to lower resource elasticity agility but lower resource costs. Conversely, selecting a smaller parameter value enhances resource elasticity agility but results in higher resource costs. Therefore, for the diverse requirements of different functions, we can adjust the specific trade-off parameter to tailor the solution to meet various needs.

6.6 Impact of Different Number of Nodes (RQ5)

To explore the impact of different cluster sizes on ComboFunc's performance, we conduct experiments with clusters configured with varying numbers of nodes, ranging from

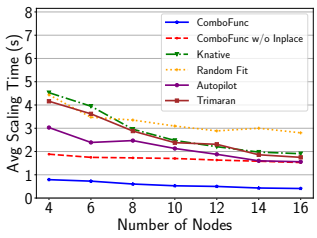


Fig. 19: Average function scaling time with a different number of nodes.

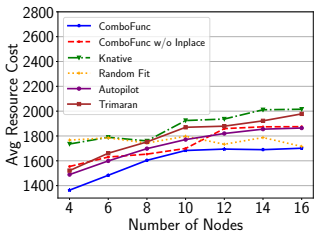


Fig. 20: Average function resource cost with a different number of nodes.

small to large. As evident from Figure 19 and Figure 20, regardless of the cluster's size, ComboFunc outperforms other strategies. An increase in the number of nodes implies that more nodes can participate in container creation, thus accelerating the scaling speed of function resources. However, for function resource cost, more nodes also mean higher resource cost overhead. ComboFunc achieves more economical and efficient function resource utilization through the combination and optimal placement of heterogeneous containers, mitigating the impact of increased resource costs associated with larger clusters.

6.7 Impact of Different Pod Creation Time (RQ6)

From the earlier analysis, we can see that the use of heterogeneous container resource combinations for resource scaling, combined with the feature of in-place pod vertical scaling, has contributed to ComboFunc's excellent resource scaling agility. Our experiments are conducted on the Knative and Kubernetes platforms, where individual container creation speeds were relatively slow, taking approximately seconds to start. However, the industry is currently developing a range of more lightweight container runtimes that can significantly reduce container creation overhead [16], [17], [50].

To investigate the resource elasticity performance of different strategies under varying speeds of individual container creation, we measure the average scaling time of containers during actual workload using a simulation platform. As shown in Figure 21, even though the container creation speed itself increases, the performance improvement of ComboFunc becomes smaller. However, ComboFunc still delivers objective resource scaling speed improvements.

Figure 22 presents the actual resource costs under different individual container creation overheads and various

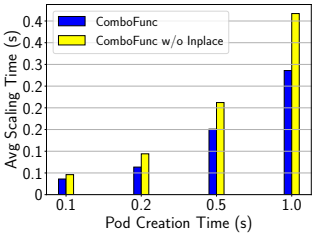


Fig. 21: Average function scaling time with different pod creation time.

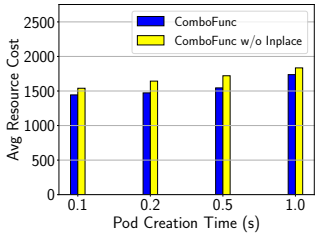


Fig. 22: Average function resource cost with different pod creation time.

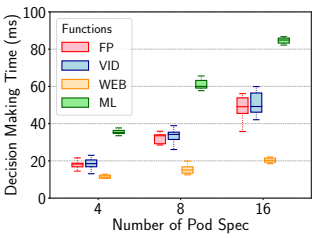


Fig. 23: Decision making time of ComboFunc under the different number of pod resource specs.

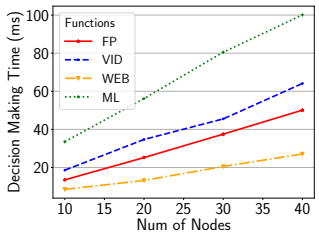


Fig. 24: Decision making time of ComboFunc under the different number of nodes.

strategies. It can be observed that the speed of creating individual containers has a relatively minor impact on the average resource costs. This is likely because the resource costs of containers are primarily determined by the resource composition strategy, with the impact of varying container creation speeds contributing less to overall resource costs.

6.8 Overhead of Solution (RQ7)

We conduct a thorough evaluation of the computational overhead of the ComboFunc. As demonstrated in Section 4.1, the computational complexity of Algorithm 1 is clearly defined. In the worst case, this complexity increases linearly with the growth of the cluster size. To show the efficiency of our algorithm, we measure the decision-making time of the ComboFunc Resource Scaler. As depicted in Figure 23, with a fixed cluster size of 16 nodes, we control the selection of varying numbers of function Pod resource specifications. Theoretically, as the number of selectable specifications for containers doubles (e.g., 4, 8, 16), the decision-making time should increase proportionally. However, as observed in the figure, the actual average decision-making time does not fully reach the theoretical doubling. This discrepancy arises because the worst-case scenario does not always occur in real systems. In our experiments, the decision-making time of ComboFunc is generally within tens of milliseconds, rarely exceeding 100 milliseconds.

Additionally, we analyze the performance of the ComboFunc algorithm across different cluster sizes. As shown in Figure 24, even in large-scale clusters, the growth in decision-making time remains within the expected linear range, without excessive increases. These results indicate that our algorithm scales effectively to larger clusters without imposing significant computational burdens. In most situations, ComboFunc efficiently optimizes the resource allocation process, and compared to the second-level cycles typically associated with container resource elasticity decisions, such decision-making times are negligible.

TABLE 4: Average decision-making time of different baselines.

Method	Latency (ms)
ComboFunc	79
Knative	45
Random Fit	35
Autopilot	99
Trimaran	60

To compare the decision time costs of other baseline algorithms, we record the average time costs during the experiment in Table 4. We find that ComboFunc's average decision time is 79 ms, only lower than Autopilot's 99 ms. ComboFunc takes longer to decide because it handles more complex problems than other methods. It not only considers the combination and placement of different containers but also decides whether to perform in-place vertical scaling for each container. This strategy, although it increases decision time, greatly improves resource efficiency. For the cloud platform, this cost is reasonable and brings significant benefits.

7 Related Work

Container scheduling in serverless. Container deployment in productions must satisfy a multitude of constraints, such as hardware requirements, fault tolerance, and resource competition [48], [51]. Current cloud resource management research primarily focuses on optimizing service placement, considering a set of containers with specific resource needs to be positioned across various nodes in a cluster [25], [26], [52], aiming to optimize resource utilization efficiency, enhance performance and reduce cost [53]. Particularly, some studies take into account factors like data transmission affinity [23], [24], disk IO [54], container image sharing [25], [26], and performance interference [27], to optimize the operational efficiency of serverless tasks, catering to their specific characteristics and requirements. Traditional container scheduling research mainly addresses bin packing or integer linear programming problems [25], [26], [55], [56]. However, ComboFunc introduces a new problem, which is to determine the combination and placement of heterogeneous container resources simultaneously, and it designs a heuristic solution based on a greedy algorithm. The mechanism is simple and efficient, suitable for rapid decision-making in real-world system environments. Some studies have achieved significant results by using deep reinforcement learning to optimize traditional complex scheduling issues [48], [57], [58], [59]. Future work could consider adopting deep learning approaches to solve these problems.

Horizontal and vertical scaling. Recent studies have also focused on elastic scaling in both horizontal and vertical dimensions. Some works have designed elastic strategies for traditional VM scenarios in both these dimensions [60]. In the context of container clouds, research has been conducted on how to optimize application performance by

combining horizontal scaling (increasing or decreasing the number of application instances) and vertical scaling (adjusting the computational resources allocated to each application instance) [13], [32], [33], [61], [62]. AWARE [32] is dedicated to solving the problem of automatic workload scaling in cloud systems. This framework considers multi-dimensional scaling (horizontal and vertical) and uses a reinforcement learning-based agent to dynamically set resource limits and scale workloads. Autopilot [13] achieves resource allocation by adjusting the number of concurrent tasks and the CPU/memory limits of individual tasks, realizing horizontal and vertical scaling of container platforms. It also utilizes machine learning algorithms based on historical data and heuristic algorithms to minimize the slackness of resource utilization and avoid issues of memory insufficiency or performance degradation. Golgi [33] reduces the cost of resource allocation by smartly over-committing functions while meeting their latency requirements. It uses nine low-level indicators to dynamically capture the runtime performance of functions, employs the Mondrian Forest classification model for performance prediction, and implements a conservative exploration-exploitation strategy for routing requests. Additionally, it executes vertical scaling to dynamically adjust the concurrency of over-committed instances, thereby maximizing request throughput and enhancing system robustness against prediction errors.

Additionally, an important research project related to ComboFunc is Escra [29], which implements dynamic resource adjustment through event-based fine-grained resource allocation. Unlike ComboFunc, Escra operates primarily at the kernel level, focusing on rapidly responsive CPU and memory resource allocation. In contrast, ComboFunc concentrates on serverless computing scenarios, emphasizing the importance of resource elasticity and scheduling strategies. ComboFunc's approach includes not only dynamic adjustment of resource scale but also optimization of container placement and flexibility in resource allocation, aimed at enhancing resource utilization and response speed, thereby better adapting to the rapid changes and resource demands in serverless environments.

However, these systems do not fully consider the heterogeneous configurations of function services, meaning that a service can correspond to multiple sets of functions with different specifications. This approach allows for a balance between elasticity speed and resource utilization. Similarly, these systems do not specifically account for the limitations in the expansion capabilities of different nodes, such as variations in expansion speed among nodes, which may result in bottlenecks during container creation. Furthermore, neither AWARE [32] nor Golgi [33] address the in-place pod vertical scaling feature of Kubernetes, a novel concept that requires targeted optimization. These limitations present opportunities for improvement within ComboFunc. The projects most similar to ComboFunc, are Autopilot [13] and Owl [18]. Autopilot implements horizontal and vertical scaling of container platforms for resource allocation, while Owl primarily focuses on optimizing container overcommitment while providing performance guarantees. In contrast, ComboFunc adopts heterogeneous container resource scheduling and optimizes the scalability of FaaS providers, representing an orthogonal improvement.

Accelerate function cold start. Many studies are dedicated to reducing the overhead of function cold starts, focusing on minimizing the additional burden during these starts [16], [17], [50], [63], [64], [65], [66], [67], [68], [69]. Firecracker [16] is a lightweight virtualization runtime developed by AWS for AWS Lambda functions, providing container-like efficiency with secure isolation. RunD [17] is a secure sandbox environment that simplifies container runtimes for faster scaling. Additionally, SAND [70] reduces data localization and function startup costs by sharing container runtimes. SOCK [71] minimizes function startup delay by caching preloaded Python containers. Pagurus [64] observes that user-operated containers often share many packages. By utilizing similar hot containers from other operations, lengthy cold starts can be eliminated. Other studies design specific container keep-alive strategies based on function request analysis to prevent cold starts [9], [37], [46], [72], [73], [74]. Some focus on hybrid methods, combining instance reservation and container elasticity for more economical resource allocation [75]. These works complement each other to enhance the efficiency of FaaS providers.

8 Limitation and Future Work

In this section, we analyze the limitations of ComboFunc and look forward to future work.

First, when scheduling containers, ComboFunc currently considers only the CPU and memory specifications of the containers. However, besides these two main factors, other factors can also affect resource scheduling. For example, a container's network bandwidth is critical, especially when dealing with large data transfers. Insufficient bandwidth can lead to increased packet delays, thus affecting the application's performance. Additionally, containers may encounter CPU cache competition, which can significantly reduce performance. Furthermore, the NUMA topology of memory is also an important consideration, as different memory access patterns can lead to variations in performance. Therefore, to improve resource utilization and optimize performance, in future work, we need to develop a more comprehensive resource model that considers these additional factors.

Second, ComboFunc has designed a function resource combination strategy based on heterogeneous containers. Customers can choose larger containers with higher concurrency or smaller containers with lower concurrency. However, there is a performance variation between containers of different specifications. To address this challenge, ComboFunc's load-balancing strategy strives to balance the workload among these containers, ensuring similar performance among different containers. However, ComboFunc lacks optimization strategies for specific function Service Level Objectives (SLOs), which may result in inadequate service quality in applications with strict performance requirements. In future work, we propose introducing more refined service quality management mechanisms. Machine learning algorithms could be used to predict the performance of different containers under various workloads, making resource scheduling more intelligent. This will allow ComboFunc not only to balance the workload among heterogeneous containers but also to optimize the execution efficiency and service quality for specific functions.

Lastly, although ComboFunc currently schedules resources based on each container's requested resources, it does not consider whether the containers are fully utilizing all the resources allocated to them. This strategy may seem sufficient for cloud platforms, as it fulfills their revenue objectives by allocating node resources. However, from the perspective of actual resource utilization efficiency, this method may lead to resource wastage, especially when containers do not fully utilize their allocated resources. For example, a container might consume a significant amount of CPU and memory resources due to over-provisioning, while these resources remain idle most of the time. In future work, we explore how to implement a dynamic overcommitment mechanism, which allows the system to overallocate physical resources while ensuring service quality, thus maximizing resource utilization efficiency.

9 Conclusion

This paper proposes ComboFunc, a system for scaling function resources for serverless platforms. Leveraging Kubernetes' in-place vertical scaling, ComboFunc employs heterogeneous containers and parallel container creation across different nodes for efficient resource scaling and improved utilization. A cost-based greedy algorithm addresses the optimization problem of resource combination and placement. Experiments on the Knative platform demonstrate that ComboFunc surpasses existing methods, increasing average resource scaling speed by up to 3.01 $\times$ and decreasing function resource cost by up to 42.6%, significantly enhancing cluster resource utilization.

References

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[2] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. of USENIX ATC*, 2018, pp. 133–146.

[3] "Amazon AWS Lambda," https://aws.amazon.com/lambda, 2022.

[4] F. Liu and Y. Niu, "Demystifying the cost of serverless computing: Towards a win-win deal," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[5] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *Proc. of USENIX ATC*, 2019, pp. 475–488.

[6] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proc. of USENIX NSDI*, 2019, pp. 193–206.

[7] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim *et al.*, "Dorylus: Affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads," in *Proc. of USENIX OSDI*, 2021, pp. 495–514.

[8] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proc. of ACM SoCC*, 2017, pp. 445–451.

[9] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Infless: a native serverless system for low-latency, high-throughput inference," in *Proc. of ACM ASPLOS*, 2022, pp. 768–781.

[10] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, "lambda dnn: Achieving predictable distributed dnn training with serverless architectures," *IEEE Transactions on Computers*, 2021.

[11] A. Mampage, S. Karunasekera, and R. Buyya, "A holistic view on resource management in serverless computing environments: Taxonomy and future directions," *ACM Computing Surveys (CSUR)*, 2022.

[12] Q. Chen, J. Qian, Y. Che, Z. Lin, J. Wang, J. Zhou, L. Song, Y. Liang, J. Wu, W. Zheng, W. Liu, L. Li, F. Liu, and K. Tan, "Yuanrong: A production general-purpose serverless system for distributed applications in the cloud," in *Proc. of ACM SIGCOMM*, 2024, pp. 843–859.

[13] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at google," in *Proc. of ACM EuroSys*, 2020, pp. 1–16.

[14] Q. Pei, Y. Yuan, H. Hu, Q. Chen, and F. Liu, "Asyfunc: A high-performance and resource-efficient serverless inference system via asymmetric functions," in *Proc. of ACM SoCC*, 2023, pp. 324–340.

[15] H. Hu, F. Liu, Q. Pei, Y. Yuan, Z. Xu, and L. Wang, "λgrapher: A resource-efficient serverless system for gnn serving through graph sharing," in *Proc. of ACM WWW*, 2024, pp. 2826–2835.

[16] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *Proc. of USENIX OSDI*, 2020, pp. 419–434.

[17] Z. Li, J. Cheng, Q. Chen, E. Guan, Z. Bian, Y. Tao, B. Zha, Q. Wang, W. Han, and M. Guo, "{RunD}: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing," in *Proc. of USENIX ATC*, 2022, pp. 53–68.

[18] H. Tian, S. Li, A. Wang, W. Wang, T. Wu, and H. Yang, "Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud," in *Proc. of ACM SoCC*, 2022, pp. 78–93.

[19] S. Li, W. Wang, J. Yang, G. Chen, and D. Lu, "Golgi: Performance-aware, resource-efficient function scheduling for serverless computing," in *Proc. of ACM SoCC*, 2023, pp. 32–47.

[20] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *Proc. of IEEE INFOCOM*. IEEE, 2022, pp. 1069–1078.

[21] Q. Weng, L. Yang, Y. Yu, W. Wang, X. Tang, G. Yang, and L. Zhang, "Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent," in *Proc. of USENIX ATC*, 2023, pp. 995–1008.

[22] Y. Zhang, Í. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *Proc. of ACM SOSP*, 2021, pp. 724–739.

[23] 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 *Proc. of ACM SoCC*, 2020, pp. 1–15.

[24] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *Proc. of USENIX NSDI*, Apr. 2023, pp. 1489–1504.

[25] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, "Exploring layered container structure for cost efficient microservice deployment," in *Proc. of IEEE INFOCOM*, 2021, pp. 1–9.

[26] D. Zeng, H. Geng, L. Gu, and Z. Li, "Layered structure aware dependent microservice placement toward cost efficient edge clouds," in *Proc. of IEEE INFOCOM*. IEEE, 2023.

[27] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li, "Understanding, predicting and scheduling serverless workloads under partial interference," in *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[28] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, "With great freedom comes great opportunity: Rethinking resource allocation for serverless functions," in *Proc. of ACM EuroSys*, 2023, pp. 381–397.

[29] G. Cusack and M. Nazari, "Escra: Event-driven, sub-second container resource allocation," in *Proc. of IEEE ICDCS*, 2022.

[30] V. Hsieh and J. Chou, "Towards serverless optimization with in-place scaling," *arXiv preprint arXiv:2311.09526*, 2023.

[31] "In-place Update of Pod Resources - Kubernetes," https://github.com/kubernetes/enhancements/tree/master/keps/sig-node/1287-in-place-update-pod-resources.

[32] H. Qiu, W. Mao, C. Wang, H. Franke, A. Youssef, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "{AWARE}: Automate workload autoscaling with reinforcement learning in production cloud systems," in *Proc. of USENIX ATC*, 2023, pp. 387–402.

[33] S. Li, W. Wang, J. Yang, G. Chen, and D. Lu, "Golgi: Performance-aware, resource-efficient function scheduling for serverless computing," in *Proc. of ACM SoCC*, 2023, pp. 32–47.
[34] P. Singh, P. Gupta, K. Jyoti, and A. Nayyar, "Research on auto-scaling of web applications in cloud: survey, trends and future directions," *Scalable Computing: Practice and Experience*, vol. 20, no. 2, pp. 399–432, 2019.
[35] "Configurations of Tencent Cloud Functions," https://cloud.tencent.com/document/product/583/68734.
[36] "Instance Concurrency of Aliyun FC," https://help.aliyun.com/zh/fc/user-guide/configure-instance-concurrency.
[37] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, and C. Xu, "The power of prediction: Microservice auto scaling via workload learning," in *Proc. of ACM SoCC*, 2022.
[38] "KubeEdge," https://github.com/kubeedge/kubeedge.
[39] "Kubernetes (K8s)," https://github.com/kubernetes/kubernetes.
[40] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proc. of ACM EuroSys*, 2015, pp. 1–17.
[41] F. Liu and Y. Niu, "Demystifying the cost of serverless computing: Towards a win-win deal," *IEEE Transactions on Parallel and Distributed Systems*, 2023.
[42] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," in *Proc. of IEEE INFOCOM*. IEEE, 2022, pp. 1868–1877.
[43] J. A. Cunningham, "The use and evaluation of yield models in integrated circuit manufacturing," *IEEE Transactions on Semiconductor Manufacturing*, vol. 3, no. 2, pp. 60–71, 1990.
[44] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *Proc. of IEEE CLOUD*. IEEE, 2019, pp. 502–504.
[45] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proc. of ACM SoCC*, 2020, pp. 30–44.
[46] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. of USENIX ATC*, 2020, pp. 205–218.
[47] "Knative: Kubernetes-based platform to deploy and manage modern serverless workloads," https://knative.dev/, 2022.
[48] Y. Zhang, Y. Yu, W. Wang, Q. Chen, J. Wu, Z. Zhang, J. Zhong, T. Ding, Q. Weng, L. Yang *et al.*, "Workload consolidation in alibaba clusters: the good, the bad, and the ugly," in *Proc. of ACM SoCC*, 2022, pp. 210–225.
[49] Trimaran, "scheduler-plugins/kep/61-trimaran-real-load-aware-scheduling at master," https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/pkg/trimaran, 2021, accessed: 2024-04-22.
[50] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. of ACM ASPLOS*, 2020, pp. 467–481.
[51] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, and R. Buyya, "Machine learning-based orchestration of containers: A taxonomy and future directions," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–35, 2022.
[52] A. Mseddi, W. Jaafar, H. Elbiaze, and W. Ajib, "Joint container placement and task provisioning in dynamic fog computing," *IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 10 028–10 040, 2019.
[53] Z. Zhong and R. Buyya, "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–24, 2020.
[54] D. Li, Y. Wei, and B. Zeng, "A dynamic i/o sensing scheduling scheme in kubernetes," in *Proc. of International Conference on High Performance Compilation, Computing and Communications*, 2020, pp. 14–19.
[55] Y. Li, X. Tang, and W. Cai, "Dynamic bin packing for on-demand cloud resource allocation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 157–170, 2015.
[56] B. Balaji, C. Kakovitch, and B. Narayanaswamy, "Fireplace: Placing firecraker virtual machines with hindsight imitation," in *Proc. of Machine Learning and Systems*, vol. 3, 2021, pp. 652–663.
[57] H. Qiu, W. Mao, A. Patke, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "Reinforcement learning for resource management in multi-tenant serverless platforms," in *Proc. of the 2nd European Workshop on Machine Learning and Systems*, 2022, pp. 20–28.
[58] Y. Bao, Y. Peng, and C. Wu, "Deep learning-based job placement in distributed machine learning clusters," in *Proc. of IEEE INFOCOM*. IEEE, 2019, pp. 505–513.
[59] M. Xu, C. Song, S. Ilager, S. S. Gill, J. Zhao, K. Ye, and C. Xu, "Coscal: Multifaceted scaling of microservices with reinforcement learning," *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 3995–4009, 2022.
[60] S. Sotiriadis, N. Bessis, C. Amza, and R. Buyya, "Elastic load balancing for dynamic virtual machine reconfiguration based on vertical and horizontal scaling," *IEEE Transactions on Services Computing*, vol. 12, no. 2, pp. 319–334, 2016.
[61] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. of IEEE CLOUD*. IEEE, 2019, pp. 329–338.
[62] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano, "Kosmos: Vertical and horizontal resource autoscaling for kubernetes," in *Proc. of International Conference on Service-Oriented Computing*. Springer, 2021, pp. 821–829.
[63] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *Proc. of USENIX HotCloud*, 2019.
[64] Z. Li, Q. Chen, and M. Guo, "Pagurus: Eliminating cold startup in serverless computing with inter-action container sharing," *arXiv preprint arXiv:2108.11240*, 2021.
[65] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: skip redundant paths to make serverless fast," in *Proc. of ACM EuroSys*, 2020, pp. 1–15.
[66] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "{FaaSNet}: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *Proc. of USENIX ATC*, 2021, pp. 443–457.
[67] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proc. of Middleware*, 2020, pp. 356–370.
[68] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proc. of Middleware*, 2020, pp. 1–13.
[69] B. Feng, Z. Ding, X. Zhou, and C. Jiang, "Heterogeneity-aware proactive elastic resource allocation for serverless applications," *IEEE Transactions on Services Computing*, 2024.
[70] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "{SAND}: Towards {High-Performance} serverless computing," in *Proc. of USENIX ATC*, 2018, pp. 923–935.
[71] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with Serverless-Optimized containers," in *Proc. of USENIX ATC*, 2018, pp. 57–70.
[72] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *Proc. of ACM ASPLOS*, 2021, pp. 386–400.
[73] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: warming serverless functions better with heterogeneity," in *Proc. of ACM ASPLOS*, 2022, pp. 753–767.
[74] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *Proc. of IEEE INFOCOM*. IEEE, 2022, pp. 1069–1078.
[75] S. Mireslami, L. Rakai, M. Wang, and B. H. Far, "Dynamic cloud resource allocation considering demand uncertainty," *IEEE Transactions on Cloud Computing*, vol. 9, no. 3, pp. 981–994, 2019.

Zhaojie Wen is currently a Ph.D. student in the School of Computer Science and Technology, Huazhong University of Science and Technology, China. His research interests include serverless computing, resource allocation, and task scheduling.

Qiong Chen received his B.Eng. degree and M.Eng. degree in the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. He is currently a research staff at the Central Software Institute, Distributed LAB of Huawei. His research interests include applied machine learning and serverless computing. He received the Best Paper Award of ACM International Conference on Future Energy Systems (ACM e-Energy) in 2018.

Quanfeng Deng is currently a Ph.D. student in the School of Computer Science and Technology, Huazhong University of Science and Technology, China. His research interests include serverless computing and cloud-native networking.

Yipei Niu received his B.Eng. degree from Henan University, and Ph.D. degree from Huazhong University of Science and Technology. He is currently a research staff at the Central Software Institute, Distributed LAB of Huawei. His research interests include cloud computing, serverless computing, container networking, and FPGA acceleration.

Zhen Song is currently a Master student in the School of Computer Science and Technology, Huazhong University of Science and Technology, China. His research interests include serverless computing and WebAssembly.

Fangming Liu (S'08, M'11, SM'16) received the B.Eng. degree from the Tsinghua University, Beijing, and the Ph.D. degree from the Hong Kong University of Science and Technology, Hong Kong. He is currently a Full Professor at the Huazhong University of Science and Technology, Wuhan, China. His research interests include cloud computing and edge computing, data center and green computing, SDN/NFV/5G, and applied ML/AI. He received the National Natural Science Fund (NSFC) for Excellent Young Scholars and the National Program Special Support for Top-Notch Young Professionals. He is a recipient of the Best Paper Award of IEEE/ACM IWQoS 2019, ACM e-Energy 2018 and IEEE GLOBECOM 2011, the First Class Prize of Natural Science of the Ministry of Education in China, as well as the Second Class Prize of National Natural Science Award in China.