FaaSPR: Latency-Oriented Placement and Routing Optimization for Serverless Workflow Processing

Yunshan Jia, Chao Jin, Qing Li, Xuanzhe Liu, *Senior Member, IEEE*, and Xin Jin, *Senior Member, IEEE*

***Abstract*—Workflow processing enhances the applicability of serverless computing while retaining the characteristics of fine-grained resource management and elastic scalability. However, current serverless platforms lack targeted optimization of placement and routing strategies for workflow processing, leading to high overheads due to inter-server data transmission, instance cold starts, and function request queuing. We propose FaaSPR, a serverless scheduling system that exploits placement and routing optimizations to minimize workflow processing latency. FaaSPR groups instances with potential data transmission and proportionally distributes groups with heterogeneous instances across multiple servers, taking into account resource constraints and historical placement traces. This method addresses the issues of poor scalability and frequent instance migrations in existing solutions. Utilizing a routing algorithm based on multi-stage linear programming, FaaSPR minimizes cross-server data transmission within and between instance groups while ensuring load balancing among instances. Experiments show that, compared to the state-of-the-art solution FaaSFlow, FaaSPR decreases the average and 99th percentile tail latency by up to 68.03% and 93.46%, respectively. Additionally, reducing workflow processing latency leads to up to 46.18% decrease in resource consumption for FaaS users.**

***Index Terms*—Serverless, scheduling, placement and routing.**

I. Introduction

SERVERLESS computing is widely supported by major cloud platforms [1], [2], [3]. In serverless scenarios, developers build applications with lightweight and stateless *functions*, while serverless platforms deploy and scale out the function instances based on user traffic, liberating developers from the maintenance and management of cloud resources and function runtimes. Many serverless applications, such as video processing [4], data analytics [5], and machine learning [6], involve serverless function workflows [7], [8], [9], [10], [11] for enhanced applicability. A serverless workflow comprises a set of functions organized as a directed acyclic graph (DAG) based on data dependencies. With the collaborative orchestration of multiple functions, the serverless workflow supports complex functionalities while maintaining independent elastic scalability for each function.

Despite being convenient and efficient, serverless workflows processing faces significant overhead from three aspects. Firstly, data dependencies between functions result in transmission overhead. The performance of different transmission solutions varies greatly [12], [13]. Data transmission within the same server occurs in shared memory with negligible latency, but cross-server transfers through remote storage (*e.g.*, AWS S3 [14] and Azure Blob [15]) has significant delays [16], [17]. Second, the lightweight nature of serverless functions leads to significant cold start overhead compared to warm starts [18], and this overhead accumulates across the workflow, amplifying its impact. Finally, the uneven load among instances when handling invocations will result in high queuing time for function requests, thus slowing down the response for user invocations. Our measurements show these factors contribute 73.91%∼80.84% of total workflow processing time, consistent with previous studies [17], [19].

In serverless workflow scheduling, the placement and routing strategy determines the placement of instances within the cluster (placement) and the execution location of each function in the workflow (routing), directly impacting workflow processing overhead. However, effective solutions for this issue are currently lacking. Although some studies claim to optimize the placement and routing problem in serverless workflows [17], [20], [21], they lack a comprehensive consideration of serverless performance factors such as high concurrency, high elasticity, limited bandwidth, and cold start overhead, resulting in performance issues. As the state-of-the-art solution, FaaSFlow [17] groups and centralized places function instances with data dependencies to facilitate local data transmission. However, this grouping algorithm is unsuitable for high-concurrency serverless scenarios, as the size of the groups will easily exceed the hardware resource limits of a single server. Additionally, the group placement strategy of FaaSFlow overlooks instance migration, leading to frequent cold start and high overhead.

In this paper, we propose FaaSPR, a serverless workflow scheduler to optimize the placement and routing strategies and minimize workflow processing latency. Generating efficient placement and routing strategies is not trivial. The first challenge is the huge search space caused by the diversity

Received 20 August 2024; revised 2 January 2025; accepted 12 March 2025; approved by IEEE TRANSACTIONS ON NETWORKING Editor S. Alouf. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4500700, in part by the National Natural Science Foundation of China under Grant 62172008 and Grant 62325201, and in part by the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). *(Corresponding author: Xin Jin.)*

Yunshan Jia, Chao Jin, Xuanzhe Liu, and Xin Jin are with the School of Computer Science, Peking University, Beijing 100871, China (e-mail: jiayunshan@pku.edu.cn; chaojin@pku.edu.cn; liuxuanzhe@pku.edu.cn; xinjinpku@pku.edu.cn).

Qing Li is with the School of Computer Science, Beijing University of Posts and Telecommunications, Beijing 100876, China (e-mail: qingli@bupt.edu.cn).

Digital Object Identifier 10.1109/TON.2025.3552407

of function instances and the high concurrency of serverless environments. Each function in a serverless workflow differs in execution duration, cold start overhead, data transmission volume, and resource requirements, presenting numerous considerations and constraints for the placement and routing strategies. Additionally, the high concurrency characteristic requires the scheduler to deploy multiple instances for a single function, and the routing algorithm needs to select a unique instance for each function in every workflow processing, further expanding the search space.

Secondly, the average and tail latencies of workflow processing must be comprehensively optimized. Existing works focused on average latency optimization instead of tail latency [17], [20], [21]. However, for workflows handling real-time requests, tail latency is crucial for user experience. Previous optimizing methods target for either the average value of a single variable (*e.g.*, latency) [17], [20] or the average values of two objective variables (*e.g.*, latency and cost) in a fixed ratio [21], which is unsuitable for average and tail latency. FaaSPR must further explore the characteristics of instance placement and function execution to avoid corner cases and excessive tail latency.

To address the two challenges, FaaSPR designs a placement and routing strategy generation algorithm based on the core insight of dispersed grouping. Regarding placement strategy, FaaSPR first groups instances serving the same function and further merges them based on data transmission overhead. The merged instance groups are proportionally distributed across multiple servers, with each server capable of handling all functions within the group. The dispersing placement method alleviates the constraints of grouping scale imposed by server resource capacity, allowing data transmission to be executed locally whenever possible. When determining the placement of instance groups, FaaSPR considers both the resource requirements of the instances and their historical placement to avoid unnecessary instance migrations, thereby reducing cold start overhead.

As a complement to the placement strategy, FaaSPR employs a multi-stage linear programming (LP)-based routing algorithm to simultaneously optimize both the average and tail workflow processing latencies. Since instance placement imbalance is inevitable under cloud platforms with complex resource constraints, the routing algorithm maximizes local data transmission while aligning traffic distribution with instance placement, avoiding function request queuing caused by traffic skew. Given the instance placement and real-time user traffic, FaaSPR approximates the theoretical optimal workflow tail latency through multi-stage LP solutions and optimizes the average processing latency with a given tail latency. With optimizations to the algorithm complexity, the routing strategy generation can be completed within milliseconds, maintaining good scalability.

We implement a prototype of FaaSPR and evaluate its performance under various serverless workflow applications. Experimental results show that, compared to the state-of-the-art solution FaaSFlow, FaaSPR reduces average and 99th tail latencies by up to 68.03% and 93.46%, respectively. Due to the pay-as-you-go nature of serverless computing [22], optimizing data transmission overhead reduces function execution time and resource usage, resulting in up to 46.18% resource consumption decrease for users.

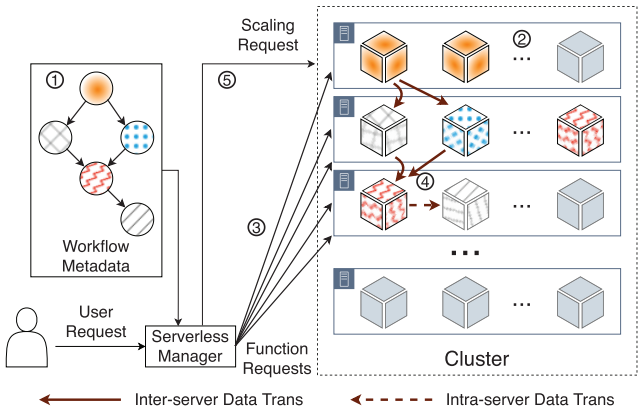


Fig. 1. Serverless workflow processing architecture.

In summary, we make the following contributions.

- We present FaaSPR, a serverless workflow scheduler that minimizes workflow processing latency.
- We propose an instance placement algorithm that maximizes local data transmission based on a horizontal and vertical grouping algorithm and minimizes instance migration to avoid cold start overhead.
- We design a multi-stage LP-based routing optimization algorithm to jointly optimize the average and tail latencies of serverless workflow processing.
- We implement a prototype of FaaSPR. Experiments show that FaaSPR reduces the average and tail latency of workflow processing by up to 68.03% and 93.46%, respectively, compared to the state-of-the-art solution FaaSFlow. The reduction in processing latency also brings up to 46.18% resource consumption savings for users.

II. Background and Motivation

This section first provides an overview of serverless workflow processing. Then, we present the optimization potential of placement and routing strategies in serverless workflow, which motivates our design.

A. Serverless Workflow Processing

Workflow processing is widely supported by existing serverless frameworks like AWS Step Functions [7], Azure Logic Apps [8], and Google Cloud Workflow [9]. Figure 1 illustrates the general architecture of serverless workflow processing, including workflow metadata storage, serverless platform manager, and worker cluster. The user-defined workflow metadata comprises the execution logic and data dependencies among all functions (①). For each function, one or more corresponding instances (generally virtual machines or containers) are deployed within the cluster to handle function requests (②). The serverless manager translates user requests into multiple function requests based on the workflow metadata and sends them to target instances (③). During execution, functions with data dependencies exchange data through remote storage services, *e.g.*, AWS S3 [14] and Azure Blob [15].

The network resources available to instances are typically limited and determined by the memory and compute resources [23], [24]. Therefore, intermediate data transmission through remote storage incurs additional overhead, especially for data-intensive workflows. Recent work allows functions executed on the same server to transfer data via shared memory [17], thus avoiding the transmission overhead. (④)

Scaling. When processing workflows, the serverless manager monitors the concurrency of each function and adjusts the deployment quantity of the corresponding instances accordingly, known as **scaling** (⑤). The formula for calculating concurrency is (average function requests per second) × (average function processing duration) [25]. The detection granularity of concurrency is typically one minute, which is also the minimum scaling interval for common serverless platforms [26], [27]. The scaling quantity for different functions within the same workflow may vary due to differences in execution duration. Serverless platforms generally deploy more instances than concurrency to ensure timely response to function requests [28], [29].

B. Motivation

Placement and routing strategies significantly influence the performance of serverless workflow processing. However, current serverless platforms lack corresponding support. The mainstream view holds that placing instances of the same workflow together can reduce communication latency [17], [20], [30], [31], [32]. However, in high-concurrency scenarios, the number of instances serving the same workflow far exceeds the resource limitations of a single server, making centralized placement unfeasible. Additionally, the scale of instances in serverless platforms is proportional to user traffic. Maintaining centralized deployment of instance groups in highly elastic scenarios can lead to unnecessary frequent instance migrations, resulting in additional cold start overhead.

Another issue with placement strategies is the lack of coordination with routing. Current cloud platform routing strategies focus on load balancing for individual functions without considering the workflow structure or instance placement. This prevents routing strategies from fully utilizing the potential of local transmission, leading to uneven load distribution among instances and causing function request queuing. Achieving optimized coordination between placement and routing strategies is not trivial. We demonstrate the performance limitations of two straightforward routing strategies in Figure 2. In this figure, instances marked with zigzag lines and dots, which have data dependencies, are deployed on servers A and B. Figure 2a illustrates the random routing strategy, where instances marked with dots randomly select a zigzag-line instance as their successor. The problem with random routing is evident as it causes unnecessary inter-server data transmission. However, simply maximizing local transmission results in overcorrection. As shown in Figure 2b, relying solely on local transmission leads to an imbalance in instance load across servers. The zigzag-line instance on server B receives many more requests than its load capacity, resulting in severe queuing delays.

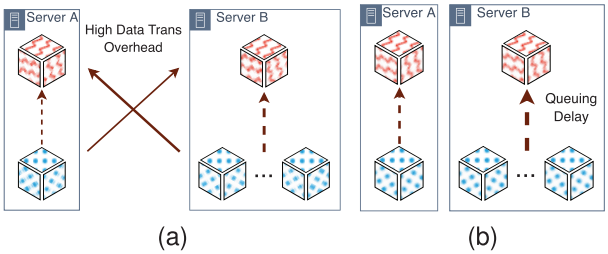


Fig. 2. Latency overhead caused by suboptimal routing strategies, where the solidity of arrows indicates the mode of data transmission (as illustrated in Figure 1), and the thickness represents the amount of transmitted data. (a) Random routing. (b) Local-first routing.

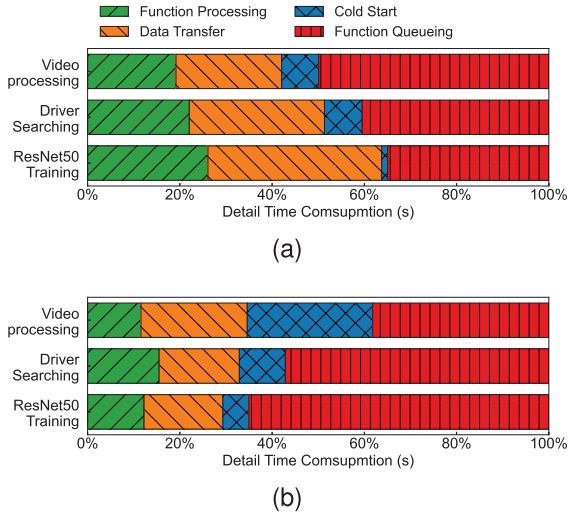


Fig. 3. Detailed time consumption of serverless workflow processing with state-of-the-art placement and routing strategies. (a) Consumption of all user requests. (b) Consumption of requests with 99th tail latency.

Figure 3 illustrates the detailed time consumption of serverless workflow processing under FaaSFlow (experimental setup details in § VII-A). It can be observed that the overhead from data transmission, cold start, and function request queuing accounts for 22.92%∼37.67%, 1.30%∼8.18%, and 34.94%∼49.90% of the average processing time, respectively. The total overhead accounts for 73.91%∼80.84% of the workflow processing time. The impact on tail latency is even more severe, with all extra overheads occupying 84.49%∼88.42% of the processing time. Moreover, the influence of cold start overhead on tail latency is significantly heightened, taking up 5.63%∼27.20% of the processing time. This result demonstrates the significant optimization potential of placement and routing strategies in serverless workflow processing.

III. FaaSPR Overview

Figure 4 illustrates the overall architecture of FaaSPR, which consists of a placement strategy generator and a routing scheduler. The placement strategy generator receives scaling requests from the serverless platform and determines the deployment location of each instance. It employs a horizontal and vertical grouping algorithm to reduce data transmission during workflow processing and avoid instance migration during scaling. The placement decisions are also sent to the routing scheduler, which uses a multi-stage LP-based

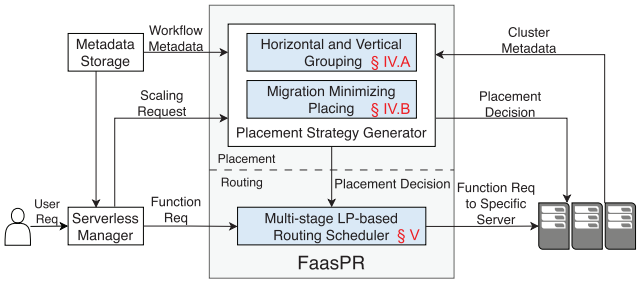


Fig. 4. FaaSPR overview.

algorithm to calculate the execution location for each function in the workflow, thereby achieving load balancing for user requests among instances while minimizing cross-server data transmission. We introduce three key techniques in FaaSPR: horizontal and vertical grouping, migration minimizing placing, and multi-stage LP-based routing optimizing.

Horizontal and Vertical Grouping. FaaSPR employs the horizontal and vertical grouping algorithm to reduce inter-server data transmission during workflow processing. Vertically, FaaSPR bundles function instances with data dependencies into instance groups. Horizontally, FaaSPR divides vertical instance groups into smaller horizontal ones approximately evenly. As the most minor deployment units, horizontal groups facilitate internal local data transmission and enables flexible deployment without being limited by server resource capacity.

Migration Minimizing Placing. FaaSPR decreases the migration of serverless instances during the scaling process (*i.e.*, to avoid terminating an instance and starting an identical one on another server) to minimize cold starts. When determining the deployment locations for a horizontal group, FaaSPR calculates the placement priority for each server based on the number of instances that need cold starts and the cold start duration, thereby maximizing the reuse of existing function instances and minimizing the cold start overhead.

Multi-stage LP-based Routing Optimizing. The ultimate objective of the multi-stage LP-based routing algorithm is to minimize the maximum processing duration of the workflow and subsequently minimize the average latency. To generate an optimal routing strategy, we consider various factors such as data transmission overhead between different instances, instance placement locations and quantities, and instance load capacities. We leverage three optimizations to reduce the algorithm execution time to the millisecond scale.

IV. FaaSPR Placement

FaaSPR employs a horizontal and vertical grouping algorithm to reduce inter-server data transmission and decrease function instance migrations during scaling to reduce cold start overhead. This section introduces the two techniques above and provides the complete placement algorithm.

A. Horizontal and Vertical Grouping

Optimizing data transmission overhead by instance grouping is widely adopted in serverless contexts [17], [20]. However, the bundling placement of instance groups conflicts with the flexible resource management of serverless computing, thereby limiting the effectiveness of optimization. We illustrate the optimization limitation with an example and demonstrate how FaaSPR optimizes placement strategies to unlock the potential of local transmission. Mainstream serverless platforms require the memory usage of instances to be proportional or approximately proportional to other resources (*e.g.*, CPU and network) [33], [34]. Therefore, we use memory as the representative instance resource in this example.

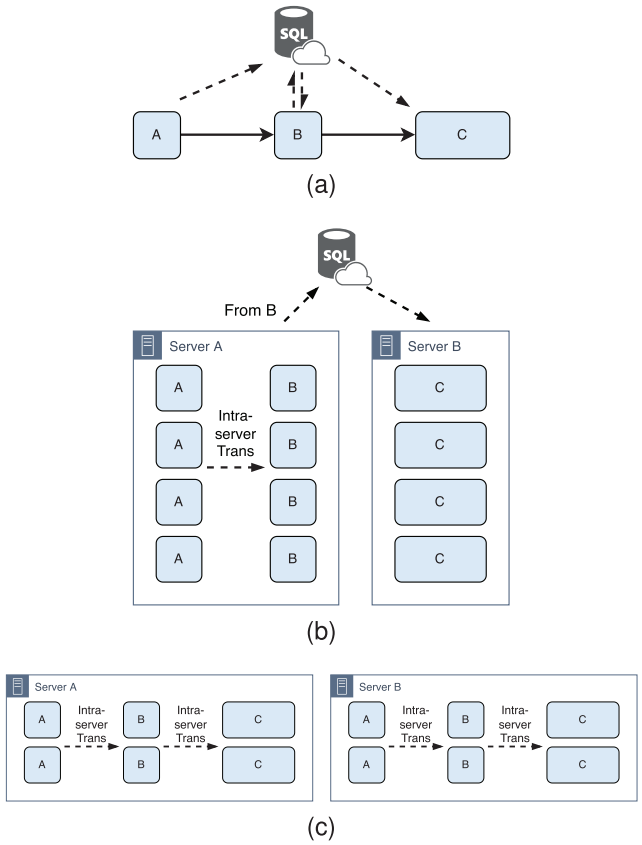


Fig. 5. Horizontal and vertical grouping example. (a) Example workflow. (b) Vertical grouping. (c) Horizontal and vertical grouping.

Figure 5a shows a sequential workflow, where functions A, B, and C interact with remote storage for data transmission. Assume that the memory requirements of instances A, B, and C are 1GB, 1GB, and 2GB, respectively. Each server has 8GB of available memory. Figure 5b illustrates the placement strategy of the state-of-the-art solution, FaaSFlow, which centrally places function instances with data dependencies.

As shown in Figure 5b, the data transmission overhead between functions A and B is eliminated. However, due to the large memory footprint, function C cannot be deployed together with others. This results in the inter-server data transmission between B and C. Similar problems are prevalent in real-world scenarios, where functions belonging to popular applications often have dozens of instances [19]. Due to the bundling resource allocation for instance groups, the grouping scale is limited by server resource capacity. FaaSPR addresses the limitation by grouping instances for both horizontal and vertical dimensions. As shown in Figure 5c, FaaSPR first consolidates the entire workflow vertically into a single group.

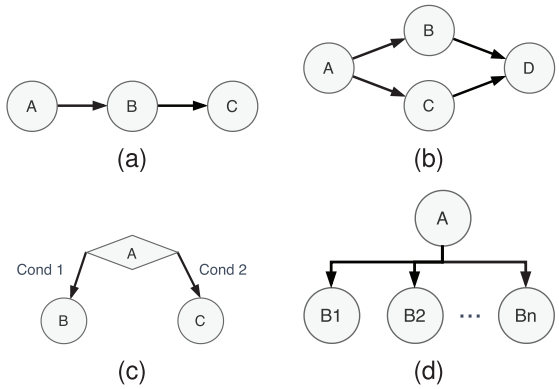


Fig. 6. Control logic in serverless workflows. (a) Sequence. (b) Parallel. (c) Switch. (d) Foreach.

Then, the vertical group is divided into two horizontal groups and placed on separate servers. Consequently, the entire workflow processing can be completed on a single server without involving remote storage.

Horizontal Grouping Rules. FaaSPR adopts an approximate even horizontal partitioning algorithm including three principles. (i) instances should be evenly distributed as much as possible among horizontal groups. Evenly grouping achieves approximate load capacities for different functions within horizontal groups, reducing the potential function request queuing caused by load capacity imbalances. (ii) Any instance in a horizontal group should have sufficient copies to avoid function request blocking caused by execution errors. (iii) FaaSPR should divide the vertical groups into as many horizontal subgroups as possible, which enables flexible deployment and avoid constraints imposed by server resource limitations.

Beyond the basic algorithm, FaaSPR employs targeted grouping strategies for specific workflow control logic to achieve more predictable performance. Figure 6 illustrates common workflow control logics in serverless platforms, including (*i*) sequence: each function has one predecessor and one successor, and all functions are executed sequentially; (*ii*) parallel: functions have multiple successors and they will execute simultaneously when dependencies are met; (*iii*) switch: functions choose specific successors based on user input or return values; and (*iv*) foreach: functions determine the number of successors based on input data, where each successor performs the same functionality but processes different inputs.

The horizontal grouping algorithm works well on sequence, parallel, and switch structures. However, the foreach structure involves concurrent function execution. When the instance capacity of a horizontal group is insufficient to support the parallel execution of all function requests, some of them will queue or be transferred to other servers, resulting in extra overhead. Additionally, the degree of concurrency of the foreach nodes depends on user input rather than a fixed value. Therefore, FaaSPR imposes additional constraints on horizontal groups containing foreach nodes. It requires that the instance numbers of foreach nodes within a horizontal group cannot be less than their maximum concurrency in the previous scaling interval. This ensures that the workflow can be processed within a single horizontal group without additional overhead.

Algorithm 1 Horizontal and Vertical Grouping

Data: vertical grouping plan vG_plan.
Input: workflow DAG *Graph*, function info f_1, f_2,...f_n.
Output: instance placement strategy P.

1: $vG\_plan \leftarrow \{\{f_1\},\{f_2\},\ldots,\{f_n\}\}$
2: $merge\_flag \leftarrow True$
3: **while** $merge\_flag$ **do**
4: $merge\_flag \leftarrow False$
5: $critical\_path \leftarrow CriticalPath(Graph, vG\_plan)$
6: $critical\_edges \leftarrow Edges(critical\_path)$
7: $list\_e \leftarrow ReversedSort(critical\_edges)$
8: **for** edge e **in** $list\_e$ **do**
9: $f_a, f_b \leftarrow From(e), To(e)$
10: $vg_a, vg_b \leftarrow vGroup(f_a), vGroup(f_b)$
11: **if** $vg_a = vg_b$ **then**
12: continue
13: $vG\_temp \leftarrow vG\_plan - \{vg_a, vg_b\}$
14: $vG\_temp \leftarrow vG\_temp \cup \{Merge(vg_a, vg_b)\}$
15: **if** $GetPlaceStrategy(vG\_temp) \neq -1$ **then**
16: $vG\_plan \leftarrow vG\_temp$
17: $merge\_flag \leftarrow True$, break
18: **Return** $GetPlaceStrategy(vG\_plan)$
19: **GetPlaceStrategy** (vG)
20: **for** P_{ij} **in** P **do** $P_{ij} \leftarrow 0$
21: **for** vg **in** vG **do**
22: $h\_num \leftarrow \min\left(\left\{\frac{f.num}{HG\_limit}\ for\ f \in vg\right\}\right)$
23: $h\_num \leftarrow \min(h\_num, server\_num)$
24: **if** $ContainForeach(vg)$ **then**
25: $h\_num \leftarrow \min(h\_num, ForeachLimit(vg))$
26: $hG \leftarrow ApproximateDivide(vg, hg\_num)$
27: **for** hg **in** hG **do**
28: **for each** server *j* **do**
29: $priority_j \leftarrow GetPriority(j, hg)$
30: **if** $\max(priority) < 0$ **then Return** -1
31: $j^* \leftarrow ServerIndex(\max(priority))$
32: $PlaceGroup(P, hg, j^*)$
33: **Return** P

Algorithm. FaaSPR updates the horizontal and vertical grouping strategy each time the serverless platform sends a scaling request. We show the key steps in Algorithm 1. The workflow DAG in the algorithm is represented by *Graph*, whose nodes f_1, f_2,...f_n represent the functions. The weights of DAG edges refer to the data transmission overhead between functions. The transmission overhead is affected by instance grouping. Inter-group transmission overhead is determined by the data volume and bandwidth, while intra-group transmission overhead is ignored. Due to varying data processing volumes in different workflow invocations, we calculate the edge weights based on the average data size of the last 1000 invocations. The algorithm outputs a placement strategy containing the deployment location for each instance.

First, FaaSPR initializes instances corresponding to each function as independent vertical groups (Line 1). The vertical groups are gradually merged in following iterations. Since the

workflow processing latency is determined by the critical path, FaaSPR adopts a *greedy grouping* strategy to simplify computational complexity. In each iteration, the algorithm prioritizes optimizing the maximum data transmission overhead on the critical path. Specifically, the algorithm calculates the DAG critical path (Line 5), extracts the edges on the critical path (Line 6), and sorts the edges in descending weight order (Line 7). Then, it traverses the edges in $list\_e$ from front to back, attempting to eliminate the edge weight by instance grouping.

When traversing edge *e*, the algorithm searches for the source/destination functions of *e* with $From()$/$To()$ (Line 9) and locates the corresponding vertical groups with $vGroup()$ (Line 10). FaaSPR attempts to merge groups if the two functions belong to different vertical groups (Lines $11 \sim 14$). If the merged grouping strategy results in a feasible placement strategy (Line 15), the algorithm updates the grouping record and enters the next iteration (Lines $16 \sim 17$). Otherwise, FaaSPR continues traversing $list\_e$. The algorithm converges when no feasible grouping is found in $list\_e$ (Line 18).

FaaSPR generates instance placement strategy *P* for vertical grouping strategies with *GetPlaceStrategy* method (Line 19). Element P_{ij} in *P* represents the number of instances for function *i* placed on server *j*, initialized to 0 (Line 20). For each vertical group *vg*, FaaSPR first determines the number of horizontal groups based on three principles. (i) The instance number of each function in a horizontal group is at least HG_limit. We empirically set HG_limit to 4 to balance grouping flexibility and performance stability (Line 22). (ii) The number of horizontal groups does not exceed the number of worker servers (Line 23). (iii) Requirement with foreach structure is satisfied (Lines $24 \sim 25$).

After horizontal grouping (Line 26), the algorithm calculates the placement priority of each horizontal group on all servers (Lines $27 \sim 29$). We leave the details of priority calculation to Algorithm 2. It is sufficient to know that a negative priority indicates infeasible placement due to resource constraints. Horizontal groups are placed on the server with the highest priority (Line $31 \sim 32$). If any horizontal group has no server available for placement, *GetPlaceStrategy* terminates and returns -1 (Line 30). Once all horizontal groups are successfully placed, the algorithm returns the final deployment plan *P* (Line 33).

B. Migration Minimizing Placing

Cold start overhead accounts for a considerable proportion of the workflow processing. When selecting the placement for instance groups, FaaSPR comprehensively considers the workflow processing overhead from both cold start and data transmission, reusing existing instances as much as possible to avoid frequent cold starts.

Algorithm 2 demonstrates how FaaSPR calculates the priority of placing horizontal group *hg* on server *j*. The algorithm first calculates the resource required (*e.g.*, memory and CPU) for placing *hg*. If the remaining idle resources on server *j* are insufficient to deploy *hg*, it returns -1 and exits (Lines $1 \sim 3$). Otherwise, the algorithm calculates and accumulates the priority for each function f in *hg*. The priority of functions comes from two aspects. First, FaaSPR calculates the proportion of reused instances when placing, from which it derives the expected saved cold start overhead as the first part of the priority (Lines $6 \sim 12$). Second, FaaSPR checks for the adjacent function of f from other groups on server *j*. Since these functions can transmit intermediate data with f locally, we accumulate the expected saved transmission overhead as the second part of the priority. For simplicity, we assume a random routing strategy in the algorithm, *i.e.*, each function request is executed on a randomly selected instance in the cluster (Lines $13 \sim 20$). Additionally, if other instances serving f are already deployed on server *j*, we slightly increase the placement priority. This is done to simplify the computational complexity of the routing strategy generation algorithm (see details in § V). The value of $Aggre\_factor$ is small enough to ensure that it only affects placement decisions when all other conditions are the same (Lines 20).

Algorithm 2 Placing Priority Calculating (GetPriority)

Input: server id *j*, available resource of server j R_j, generating placement strategy *P*, placement strategy of last scaling interval $last\_P$ and horizontal group to place *hg*.
Output: placing priority p.

1: $m\_need \leftarrow \sum_{f \in hg} f.resource * \frac{f.num}{hg\_num}$
2: **if** $ResourceOccupied(P, j) + m\_need > R_j$ **then**
3: **Return** -1
4: $p \leftarrow 0$
5: **for** $f \in hg$ **do**
6: $reuse\_cap \leftarrow instanceNum(last\_P, j, f)$
7: $occ\_num \leftarrow instanceNum(P, j, f)$
8: **if** $reuse\_cap > occ\_num$ **then**
9: $reuse\_num \leftarrow reuse\_cap - occ\_num$
10: $reuse\_num \leftarrow \min\left(reuse\_num, \frac{f.num}{hg\_num}\right)$
11: $reuse\_ratio \leftarrow \frac{hg\_num * reuse\_num}{f.num}$
12: $p \leftarrow p + reuse\_ratio * f.cs\_time$
13: **for** $e \in IncomingEdge(f)$ **do**
14: $f^* \leftarrow From(e)$
15: **if** $f^* \notin hg$ **then**
16: $p \leftarrow p + \frac{instanceNum(P,j,f^*)}{f^*.num} * e.weight$
17: **for** $e \in OutgoingEdge(f)$ **do**
18: $f^* \leftarrow To(e)$
19: **if** $f^* \notin hg$ **then**
20: $p \leftarrow p + \frac{instanceNum(P,j,f^*)}{f^*.num} * e.weight$
21: $p \leftarrow p + instanceNum(P, j, f) * Aggre\_factor$
22: **Return** p

Multiple workflows. FaaSPR is also applicable to handling multiple workflows simultaneously. In this case, the placement algorithm is applied to each workflow sequentially. Once the placement strategy for a workflow is determined, FaaSPR updates the available resource capacity on all worker servers and handles the next workflow. To avoid performance skew caused by placement order, FaaSPR randomizes workflow placement order for each scaling request.

V. Multi-Stage LP-Based Routing in FaaSPR

FaaSPR adopts a multi-stage LP-based routing algorithm to specify the execution location for each function request. The

TABLE I
NOTATION

Notation	Description
n	Number of functions in the workflow
m	Number of servers in the cluster
$t_i, i \in [1, n]$	Execution duration of function i
$t_{scaling}$	Time interval between scaling requests
$P_{ij}, i \in [1, n]$, $j \in [1, m]$	Number of instances of function i placed on server j
w	Number of branches in workflow
$O_i, i \in [1, w]$	Invocation frequency for branch i
r	Number of routing paths
$R_{ijk}, i \in [1, n]$, $j \in [1, m], k \in [1, r]$	If workflow executes function i on server j in routing path k
$B_{ij}, i \in [1, w]$, $j \in [1, r]$	If routing path j is bound with branch i
$l_i, i \in [1, r]$	Workflow response latency of routing path i
$o_i, i \in [1, r]$	Execution frequency of routing path i

algorithm uses the LP model to minimize the average processing latency of the workflow. Meanwhile, we incorporate an additional latency constraint into the LP model to control the tail latency. By minimizing the latency constraint through multiple LP solutions, the algorithm optimizes both average and tail latency for workflow processing. In this section, we introduce the routing algorithm in detail and explain how to keep the algorithm overhead within milliseconds to ensure scalability.

This section describes the routing algorithm for a single workflow. When handling multiple workflows, the scheduler applies the routing algorithm to each workflow in parallel. The routing strategies among multiple workflows do not interfere with each other.

A. System Model

In this section, we assume that the scheduler generates a routing strategy for a workflow consisting of *n* functions in a serverless cluster with *m* worker servers. The execution durations of the functions are denoted as $t_1, t_2, \ldots, t_n$. The instance scaling interval of the serverless platform is $t_{scaling}$. Matrix *P* refers to the placement of function instances, where P_{ij} indicates the number of instances of function *i* on server *j*. To support switch structures in serverless workflows, we assume that the workflow has *w* branches. For workflows without any switch nodes, $w = 1$. Each branch corresponds to a unique set of functions, and each workflow invocation executes a specific branch. The invocation frequency of each branch during the next scaling interval is denoted as $O_1, O_2, \ldots, O_w$. The invocation frequency is one of the inputs to our algorithm. In the current version, we predict the future invocation frequency based on the historical trace and instance number. More accurate prediction algorithms, which could further enhance algorithm performance, are beyond the scope of this paper.

Each time the serverless platform issues a scaling request, the placement algorithm is executed first to generate a placement strategy *P*. Thereafter, the routing algorithm determines the execution locations for function requests based on the specific *P*. The correspondence between function requests and execution servers is referred to as the *routing path*. For example, for the workflow segment and instance placement shown in Figure 2a, all feasible routing paths are listed in Table II.

TABLE II
ROUTING PATHS FOR FIGURE 2

Routing Path	Execution location for green function	Execution location for blue function
1	Server A	Server A
2	Server B	Server A
3	Server A	Server B
4	Server B	Server B

For a given placement strategy, each branch of the workflow corresponds to one or more feasible routing paths. We denote the total number of routing paths from all the branches as *r*. Matrix *B* represents the affiliation between routing paths and branches. $B_{ij} = 1$ indicates that routing path *j* is affiliated with branch *i*; otherwise, $B_{ij} = 0$. Routing path information is recorded in matrix *R*. Each element of *R* is either 1 or 0. $R_{ijk} = 1$ means function *i* is executed on server *j* in routing path *k*; otherwise, $R_{ijk} = 0$. If a branch in the workflow does not execute function *i*, then for any paths *j* belonging to this branch, we have $\sum_{j=1}^{m} R_{ijk} = 0$. Since the execution locations have been determined in routing paths, the data transmission overhead and overall workflow latency can be calculated. $l_1, l_2, \ldots, l_r$ denote the workflow processing latency for each routing path. We do not involve cold starts here as they affects only a small number of requests at the beginning of each scaling interval. $o_1, o_2, \ldots, o_r$ represent the execution frequency of each routing path, which is the output of FaaSPR routing algorithm.

Table I summarizes the used symbols.

B. LP Constraints

FaaSPR leverages the LP model to optimize the execution frequencies of routing paths. The LP model is subject to three constraints: execution frequency, instance capacity, and latency threshold.

Execution Frequency Constraint. Since each workflow invocation corresponds to a routing path, the total execution frequency of all routing paths is at least the overall access frequency of the workflow. Furthermore, for each branch of the workflow, the sum of the execution frequencies of its corresponding routing paths is no less than the access frequency of that branch. Therefore, we can formulate Constraint 1.

$$\sum_{j=1}^{r} B_{ij} o_i \geq O_i, \forall i \in [1, w] \qquad \text{(Constraint 1)}$$

Instance Capacity Constraint. To avoid long queuing, the number of function requests assigned to each server should not exceed the instance capacity. To enhance the system robustness, we multiply the load capacity of each instance by a constant factor α, to cope with traffic bursts and instance failures, $\alpha \in (0, 1]$. A larger α means full utilization of instance capacity on low-latency paths, while a smaller α allows more room for unexpected situations. We currently

use an empirical value 0.9 for α to balance performance and stability. Therefore, we can derive Constraint 2.

$$\sum_{k=1}^{r} R_{ijk} o_k \leq \frac{\alpha P_{ij}}{t_i}, \forall i \in [1,n], \forall j \in [1,m] \quad \text{(Constraint 2)}$$

Latency Threshold Constraint. FaaSPR leverages auxiliary constraints to minimize the tail latency in workflow processing. We use a latency threshold L to represent the maximum allowed latency of all paths in the final routing strategy. Based on this, we introduce Constraint 3.

$$\begin{cases} o_i \geq 0, & l_i \leq L \\ o_i = 0, & l_i > L \end{cases} \quad \forall i \in [1,r] \qquad \text{(Constraint 3)}$$

To simplify Constraint 3, we reorder all routing paths in ascending latency order, which means $l_1 \leq l_2 \leq \ldots \leq l_r$. Without loss of generality, we constrain L to be a specific element in $l_1, l_2, \ldots, l_r$, with x representing the element index, $x \in \{1, 2, \ldots, r\}$. Therefore, Constraint 3 is equivalent to Constraint 3* ($L = l_x$).

$$\begin{cases} o_i \geq 0, & \forall i \in [1,x] \\ o_i = 0, & \forall i \in (x,r] \end{cases} \qquad \text{(Constraint 3*)}$$

C. Objective Function and Final LP Model

The objective of LP models is to minimize the average latency, which can be characterized as:

$$\frac{\sum_{i=1}^{r} l_i o_i t_{scaling}}{\sum_{j=1}^{w} O_j t_{scaling}}$$

Since the invocation frequencies $O_1, O_2, \ldots, O_j$ are determined and scaling interval $t_{scaling}$ is constant, the LP objective is equivalent to $\min \sum_{i=1}^{r} l_i o_i$.

The complete LP model is presented below.

$$\begin{aligned} \min \quad & \sum_{i=1}^{r} l_i o_i \\ \text{s.t.} \quad & \sum_{j=1}^{r} B_{ij} o_j \geq O_i, \quad \forall i \in [1,w] \\ & \sum_{k=1}^{r} R_{ijk} o_k \leq \frac{\alpha P_{ij}}{t_i}, \quad \forall i \in [1,n], \ \forall j \in [1,m] \\ & o_i \geq 0, \quad \forall i \in [1,x] \\ & o_i = 0, \quad \forall i \in (x,r] \end{aligned}$$

D. Multi-Stage LP

Except for $o_1, o_2, \ldots, o_r$ and x, all variables in the LP model are already determined before the algorithm begins. With a specific x, the LP output $o_1, o_2, \ldots, o_r$ represents the execution frequencies of the routing paths which minimize average latency. At the same time, due to Constraint 3, routing paths whose latencies exceed l_x have zero execution frequency, thereby controlling the maximum workflow processing latency within l_x. As a result, FaaSPR achieves a comprehensive optimization of both average and tail latency. The objective of the algorithm is to find the smallest x and l_x that allow LP to have feasible solutions. FaaSPR determines the final x and routing strategy $o_1, o_2, \ldots, o_r$ by adjusting x and re-solving the LP model multiple times. o_i ($i \in [1,r]$) represents the frequency with which routing path *i* is selected for workflow invocations. Note that multiple paths may be selected and used simultaneously (*i.e.*, $|\{o_i | o_i > 0\}| \geq 1$).

During the scheduling phase, we select a routing path for each workflow invocation using a weighted random algorithm. For workflows without switch nodes ($w = 1$), the probability of selecting routing paths *i* is $\frac{o_i}{\sum_{j=1}^{r} o_j}$. For workflows with multiple branches, it is impossible to determine which branch will be executed at the beginning of the workflow execution. FaaSPR first randomly selects route path *i* from all the paths based on probability $\frac{o_i}{\sum_{j=1}^{r} o_j}$. When executing function *j* in the workflow, if path *i* includes the execution location of function *j*, the scheduler deploys function *j* according to path *i*. Alternatively, we search for all the routing paths containing function *j* and select a new path *k* from them. The algorithm then assigns function *j* and subsequent function requests according to path *k*.

Overhead. The algorithm overhead is negligible with small clusters. However, as the cluster size and the number of instances increase, the number of routing paths grows exponentially, slowing down LP resolution. We adopt three approaches to reduce the algorithm overhead.

First, FaaSPR groups the cluster nodes into groups of 8 servers. The scaling request is distributed to multiple server groups, and each group generating placement and routing strategies individually. Server grouping reduces the size of individual LP models and speeds up computation through parallelism. Although this method restricts cross-server data transmission to within the same server group, experiments show no impact on the algorithm performance. When distributing scaling requests, FaaSPR prioritizes server groups with the largest resource capacity and maximizes the resource utilization of these groups, avoiding resource fragmentation. Second, when generating placement strategies, FaaSPR prioritizes placing instances together when other conditions are the same (Line 21 in Algorithm 2), reducing the number of feasible routing paths as well as LP model size. Finally, FaaSPR uses binary search to determine x from $\{1, 2, \ldots, r\}$, reducing LP solution times. With these optimizations, the placement and routing algorithms maintain millisecond-level latencies even in large-scale clusters. We demonstrate the scalability of FaaSPR in Section VII-E.

VI. Implementation

We implement a prototype of FaaSPR with ∼2800 LOC in Python, with Kubernetes [35] for instance management, Redis [36] for intermediate data storage, and SciPy [37] for LP solving.

Workflow manager. We use containers as the serverless function instance and implement a workflow manager based on the Kubernetes container orchestration. The manager parses user-defined workflow from YAML files and deploys functions based on dependencies. We limit container resource usage through Kubernetes deployment configurations and specify

instance placement locations based on node affinity. Data transmission between functions is implemented using Redis. We deploy a Redis instance on each worker server for local data transmission and on a separate server as the remote storage.

Transmission manager. We develop a data transmission library for our prototype. In our prototype, function outputs are saved in local memory after execution. Before assigning function requests, the scheduler identifies the storage location of the function inputs based on deployment trace. The storage information is sent to the worker server along with function requests. During execution, functions read and write data through unified interfaces provided by the data transmission library. For write operations, the library stores data in the local Redis. When reading data, the library first checks its storage location. If the data is stored on the local server, the library retrieves it directly from the local Redis. Otherwise, the library sends a remote request to transfer the data from the source server to remote storage and subsequently retrieves it from the remote storage. The library tracks the size of transmission data and sends the size information back to the scheduler with the function completion signal. The scheduler calculates the average transmission data size according to the invocation trace, which is used for workflow placement and routing. The library can easily integrate with other remote storage services by modifying the storage access instructions.

Placement and routing scheduler. We implement the FaaSPR algorithm with ∼700 LOC in Python. In each scaling process, the scheduler first calculates the instance placement strategy based on user-provided workflow information, the current instance placement status, and the instance scaling request. Subsequently, the scheduler generates LP models based on the placement strategy and solves them with SciPy. The final model solution indicates the probabilities for routing path selection. The scheduler chooses a routing path for each workflow invocation based on the weighted random algorithm.

VII. Evaluation

In this section, we first evaluate the overall performance of FaaSPR against the state-of-the-art solution FaaSFlow [17] (§ VII-B). Second, we explore from a micro perspective whether FaaSPR can avoid latency fluctuations caused by instance scaling (§ VII-C). Then, we dive into FaaSPR to analyze the effectiveness of its techniques (§ VII-D). Finally, we investigate the scalability of FaaSPR (§ VII-E).

A. Evaluation Setup

Benchmarks. We selected six workflows as benchmarks to validate the performance of FaaSPR under different applications.

- **Video processing.** The video processing application extracts metadata, converts formats, and adds watermarks to videos using FFmpeg [38]. Depending on the size of the uploaded videos, users can choose either simple sequential or parallel processing based on automatic video segmentation. This application is accessible on Alibaba Cloud Computing as a serverless workflow use case [4].
- **Driver searching.** As part of the ride-sharing system, the driver searching function assists users in finding nearby drivers available for carpooling. Users need to log in to the system first. Upon successful login, the system retrieves data from the driver location database and filters out a list of nearby drivers for the user to choose from. This is one of the examples provided by Azure Logic Apps [5].
- **ResNet50 training.** Online model updating is one of the typical applications of serverless workflow. This application periodically fetches updated data from an online database and performs data preprocessing and model training. After that, the application evaluates the inference accuracy of the new model and returns the evaluation results along with the updated model. Relevant code can be found in the examples of AWS Step Functions [6].
- **Checklist review, calculator, and travel guide.** To demonstrate the performance of FaaSPR under millisecond-scale workflows, we extract these three workflows from the example code provided by AWS Step Functions [39], Azure Logic Apps [40], and Google Cloud Workflows [41]. The processing times of these workflows are in the millisecond range.

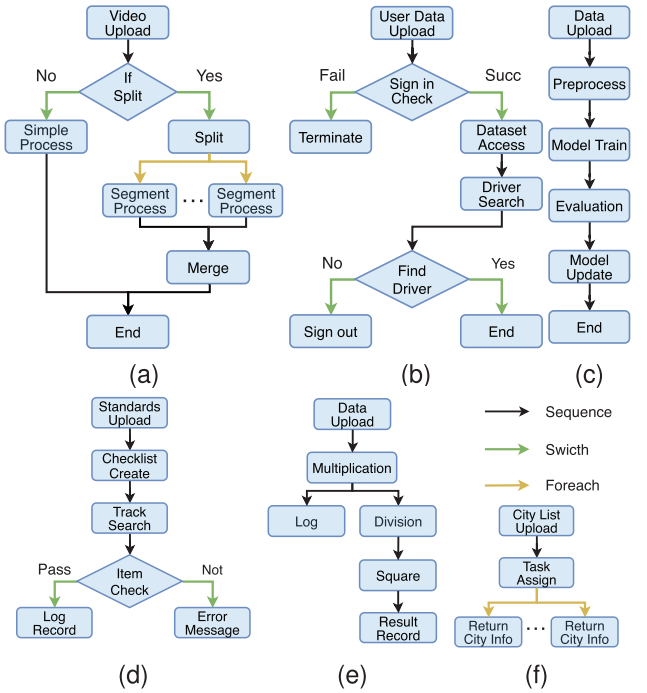


Fig. 7. Application benchmarks. (a) Video Processing. (b) Driver Searching. (c) ResNet50 Training. (d) Checklist Review. (e) Calculator. (f) Travel Guide.

Figure 7 summarizes the control logic of these workflows. The application workflows incorporate various control logics (*e.g.*, sequential, switch, and foreach) and vary in terms of task scale and data volumes, thus providing a comprehensive reflection of the actual performance of FaaSPR. In the driver searching application, we assume that one in ten users fails to log in. In other switch nodes, the probability of executing different branches is equal.

Workload generation. We generate the simulated user traffic based on the Azure Function Dataset [18]. Since this dataset does not include workflow information, we use traces of individual functions instead. Unless otherwise specified, the trace is extracted from the top 2% frequent invocation traces, with the average and peak invocation frequencies of 203.8 req/min and 264 req/min, respectively. The data selection is representative because, in serverless scenarios, the majority of requests come from a small number of applications that are invoked most frequently [18]. Since invocation traces in the dataset are recorded at minute granularity, we generate invocation patterns within each minute based on the Poisson distribution. In line with the existing platform [26], [27], we set the scaling interval to 1 minute.

We treat memory as the primary resource of an instance, with CPU resources proportional to memory (2GB per CPU core). The default available bandwidth for each instance is 50MB/s. Based on local performance testing, we set the memory requirements for each type of instance to range from 0.5GB to 2GB, which is consistent with mainstream serverless platform [33], [34].

Cluster configuration. Our experiments are conducted on CloudLab using a cluster consisting of 10 r6525 instances (64 cores, AMD EPYC 7543 CPU @ 2.8GHz). Eight of them serve as working servers (each has 64 CPU cores and 128GB memory of allocable resources), one as the remote storage, and the last as the scheduler and user request generator. To simulate the mixed workloads of different applications within a cluster, we assume that only a portion of the worker server resources is available for the target application. Unless otherwise specified, the available resource proportions across different servers are sampled from a normal distribution with a standard deviation of 0.8, with a total available resource of ∼402 GB memory.

Baseline. We compare FaaSPR against FaaSFlow, the state-of-the-art solution that systematically optimizes placement and routing strategies for serverless platforms. Based on a grouping algorithm, FaaSFlow centrality deploys functions with data dependencies to maximize local data transmission.

We compared the average and tail latency, as well as the resource consumption, between FaaSPR and the baseline. Resource consumption refers to the sum of the product of resource occupation and execution time for all instances processing the workflow, which reflects the cost for users. For ease of presentation, we normalize resource consumption based on FaaSFlow performance in all comparisons. Each data point following is obtained during an hour-long evaluation, and we excluded the latency data from the first 5 minutes to disregard the influence of the initial cold start. Unless otherwise specified, we use video processing, normalized resource distribution, and 50MB/s bandwidth as the represent setup.

B. Benefits of FaaSPR

We first compare the overall performance of FaaSPR to FaaSFlow under different workflow, bandwidth, and resource distribution configurations.

Comparison under different workflows. Figure 8 compares the performance of FaaSPR and FaaSFlow in processing different workflows. For second-level applications (*i.e.*, video processing, driver searching, and ResNet50 training), the average and tail latencies of FaaSPR are reduced by 48.62%∼62.97% and 41.64%∼51.99%, respectively, compared to FaaSFlow. Additionally, FaaSPR achieves a 23.77%∼46.18% reduction in resource usage. For millisecond-level workflows (*i.e.*, checklist review, calculator, and travel guide), the average and tail latencies of FaaSPR are 16.82%∼35.91% and 86.60%∼93.46% lower than FaaSFlow, respectively. The resource consumption of FaaSPR is 1.55%∼11.07% lower than FaaSFlow.

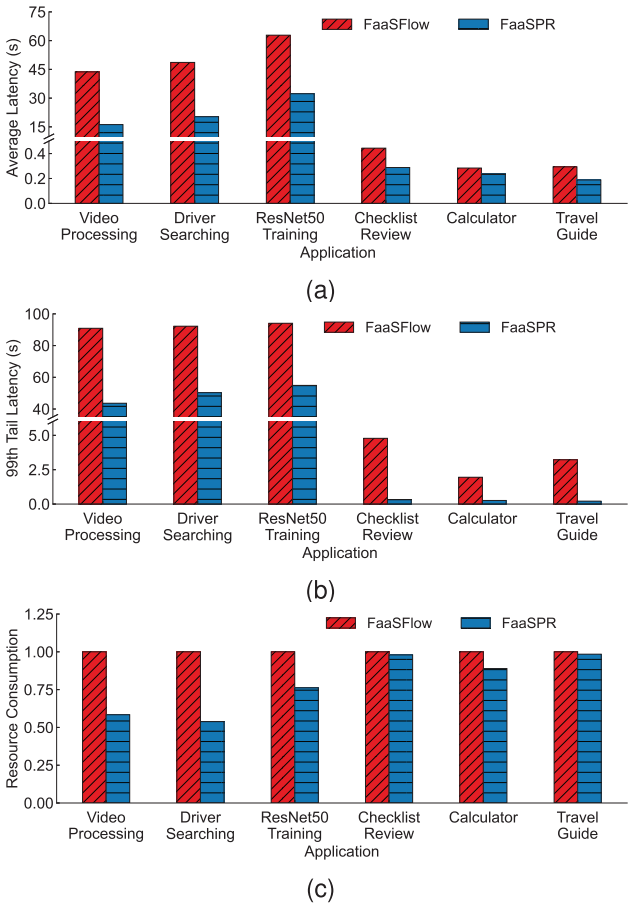


Fig. 8. Comparison between FaaSPR and FaaSFlow with different workflows. (a) Average latency. (b) 99th tail latency. (c) Resource consumption.

Compared with second-level workflows, FaaSPR significantly reduces the tail latency of millisecond-level workflows, while the improvements in average latency and resource utilization are relatively small. This is because millisecond-level workflows need fewer instances. In most cases, all instances can be deployed on a single server without the need for additional placement algorithms. Therefore, the instance placement strategies of FaaSFlow and FaaSPR are similar, with the main performance difference arising from the cold start overhead. Due to the lightweight nature of millisecond-level workflows, the impact of cold start latency on overall delay is more pronounced, making the optimization of tail latency more significant.

Comparison under different bandwidths. Secondly, we demonstrate the performance differences between FaaSPR and

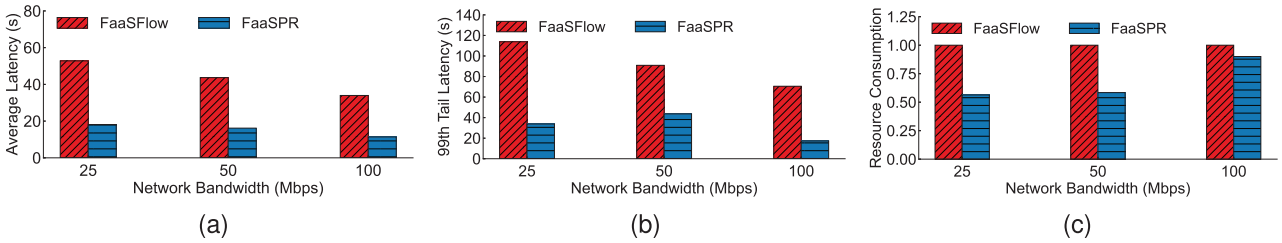


Fig. 9. Comparison between FaaSPR and FaaSFlow under different network bandwidths. (a) Average latency. (b) 99th tail latency. (c) Resource consumption.

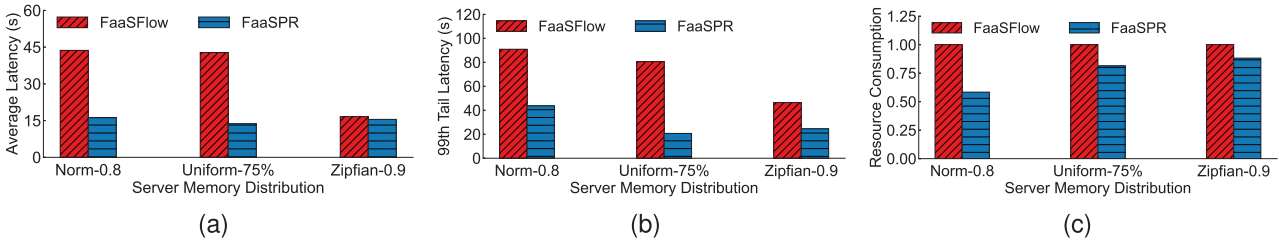


Fig. 10. Comparison between FaaSPR and FaaSFlow under different resource distributions. (a) Average latency. (b) 99th tail latency. (c) Resource consumption.

FaaSFlow in different bandwidth constraints. In Figure 9, we adjusted the bandwidth limitation between remote storage and worker servers from 25MB/s to 100MB/s. Under different bandwidth conditions, FaaSPR improves average latency, 99th tail latency, and resource consumption by 62.97%∼66.50%, 51.99%∼75.46%, and 9.94%∼43.66%, respectively, compared to FaaSFlow.

Compared to FaaSFlow, the performance of FaaSPR in tail latency is less affected by bandwidth because FaaSPR minimizes the amount of data transmission across servers. As a result, the workflow tail latency is primarily influenced by instance cold starts and function request queuing. When the bandwidth is high, the advantage of FaaSPR in terms of resource consumption diminishes. This is because FaaSPR reduces instance execution duration by minimizing data transmission overhead. When the bandwidth is high, the transmission overhead is already small, leaving less room for optimization.

Comparison under different resource distributions. We show the performance of FaaSPR under different server resource distributions. We introduce two new resource distribution patterns: uniform distribution and Zipf distribution. In the uniform distribution, 75% of the resources are available for each server to process workflows. In the Zipf distribution, the proportion of available resources on each server is determined by a Zipf distribution (with $\alpha = 0.9$ and n =8). As shown in Figure 10, the average latency, 99th tail latency, and resource consumption of FaaSPR are 6.55%∼68.03%, 47.22%∼74.33%, and 11.79%∼41.62% lower than FaaSFlow, respectively.

FaaSPR performs similarly under normal and uniform distributions. Under the Zipf distribution, where server resources are relatively concentrated, the advantage of dispersed grouping in FaaSPR over FaaSFlow is less pronounced, resulting in similar performance in terms of average latency and resource usage. In terms of tail latency, FaaSPR shows a significant advantage over FaaSFlow due to its optimization of cold start overhead and request queuing delays.

TABLE III
PERFORMANCE COMPARISON UNDER MULTIPLE WORKFLOWS

Application	Average latency		99th tail latency	
	FaaSFlow	FaaSPR	FaaSFlow	FaaSPR
Video Processing	36.84	16.03 (-56.49%)	76.24	56.73 (-25.60%)
Driver Searching	42.06	18.29 (-56.51%)	91.59	66.02 (-27.92%)
ResNet50 Training	74.24	26.63 (-64.12%)	115.54	63.68 (-44.89%)
Checklist Review	0.38	0.37 (-0.28%)	1.60	0.87 (-45.65%)
Calculator	0.31	0.28 (-0.78%)	1.51	0.52 (-65.53%)
Travel Guide	0.44	0.21 (-53.17%)	4.09	0.28 (-93.22%)

Comparison under multiple workflows. We demonstrate the performance of FaaSPR when serving multiple workflows simultaneously. In this section, we scale down the invocation frequency accordingly to ensure that the total resource demand of all instances does not exceed the cluster capacity. Table III shows the average and tail latencies of all workflows. Compared with FaaSFlow, FaaSPR reduces the average and tail latencies of different workflows by 0.28%∼64.12% and 25.60%∼93.22%, respectively. Overall, FaaSPR effectively reduces both the average and tail latencies when handling multiple workflows. In terms of resource usage, FaaSPR reduces resource consumption by up to 54.96% compared with FaaSFlow when invoking different workflows. Detailed resource usage data is omitted for space reasons.

Performance analysis. To further demonstrate the reasons for FaaSPR performance advantage, Figure 11 provides a detailed illustration of the average time consumption of FaaSPR and FaaSFlow during workflow processing. Compared to FaaSFlow, FaaSPR reduces time consumption in inter-server data transmission, instance cold starts, and function request queuing by 55.41%, 90.59%, and 84.05%, respectively. Overall, FaaSPR reduces the extra overhead in workflow processing by 76.58%.

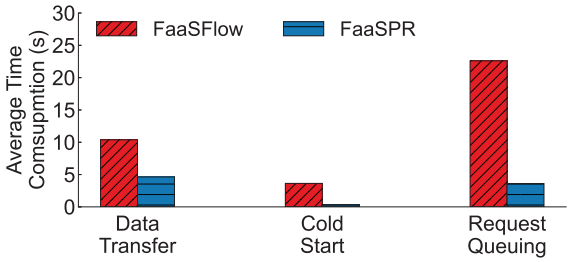


Fig. 11. Time consumption in workflow processing.

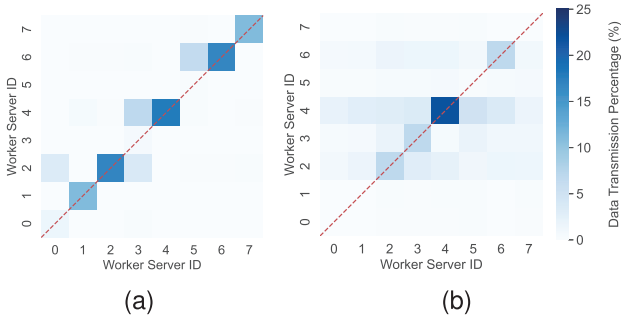


Fig. 12. Data transmission distribution for (a) FaaSPR and (b) FaaSFlow.

The outperforms of FaaSPR are attributed to its optimization for placement and routing strategies. First, FaaSPR maximizes local data transfer based on the horizontal and vertical grouping algorithm. Secondly, FaaSPR maintains the original placement of function instances as much as possible during the scaling process to minimize cold start overhead. In contrast, FaaSFlow completely disregards cold start overhead and causes frequent instance migrations. Finally, FaaSPR balances the load capacity of different functions within worker servers through the grouping algorithm and minimizes function request queuing delays through further routing optimization. In addition, due to the optimization of cold-start and data transfer overhead, FaaSPR objectively reduces the execution time of function requests and decreases potential function request queuing delays.

We give a more detailed demonstration of FaaSPR advantage in data transmission. Figure 12 illustrates the transmission distribution among worker servers under FaaSPR and FaaSFlow. The horizontal and vertical axes represent server indices, and the color depth indicates the proportion of data transmission from horizontal to vertical servers. The red lines refer to local data transmission. The proportion of local data transmission in FaaSPR and FaaSFlow is 76.20% and 42.72%, respectively. As shown in the figure, the local data transmission in FaaSFlow is more concentrated but accounts for a smaller proportion. This is because FaaSFlow tends to deploy instances with data dependencies in a concentrated manner. The scale of concentrated deployment is limited by the resource capacity of servers under high load, leading to suboptimal performance. In contrast, FaaSPR disperses instance groups across multiple servers, breaking the resource limitations of a single server and enhancing the potential for local data transmission.

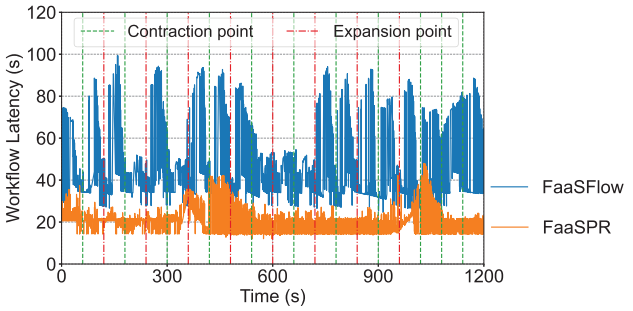


Fig. 13. Comparison between FaaSPR and FaaSFlow at the micro-scale.

C. Micro-Scale Benchmark

In this section, we show the advantage of FaaSPR during function scaling through fine-grained performance monitoring. Figure 13 shows the corresponding latency per workflow invocation for FaaSPR and FaaSFlow over 20 minutes. The figure includes 19 instances of scaling, which we categorize into two types: instance expansion due to increased user traffic (marked with red dashed lines) and instance contraction due to decreased user traffic (marked with green dashed lines). It can be observed that FaaSPR maintains stable workflow processing latencies during most scaling processes. In contrast, FaaSFlow exhibits significant latency fluctuations, with peak latencies of 99.4 seconds. Additionally, the latency of FaaSFlow remains consistently higher than FaaSPR throughout the entire scaling interval.

The performance advantage of FaaPR over FaaSFlow comes from two sources. On the one hand, FaaSPR maintains the original placement of function instances during the scaling process. This not only reduces the cold start overhead for individual functions but also avoids the cascading cold start problem, thereby avoiding significant fluctuations in workflow processing latency. Note that the cascading cold start affects not only the latencies near the scaling points but also leads to queuing and blocking of subsequent function requests, resulting in increased response latency for a large number of user invocations. On the other hand, based on the horizontal and vertical grouping algorithm and routing optimization, FaaSPR increases the proportion of local data transmission and reduces the queuing delay of function requests. This makes the workflow processing efficiency of FaaSPR steadily higher than that of FaaSPR in the scaling interval.

D. Effectiveness of FaaSPR

In this section, we investigate the impact of the techniques employed by FaaSPR. We measure FaaSPR with the following three simplified versions: (i) **DataTrans*** based on random routing strategy, utilizes horizontal and vertical grouping algorithms to reduce inter-server traffic; (ii) **ColdStart*** based on random routing strategy, utilizes migration minimizing placement to avoid cold start; (iii) **Routing*** adopts random placement and multi-stage LP-based routing strategy in FaaSPR. We also provide the evaluation results of FaaSFlow for reference. As shown in Figure 14, compared to FaaSFlow, the simplified versions DataTrans*, ColdStart*, and Routing* reduce average latency by 18.15%, 51.42%,

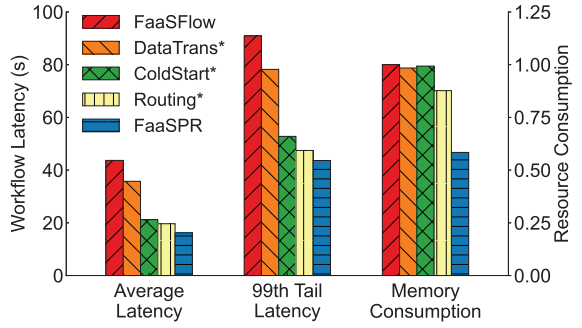


Fig. 14. Latency comparison between FaaSPR and three simplified versions.

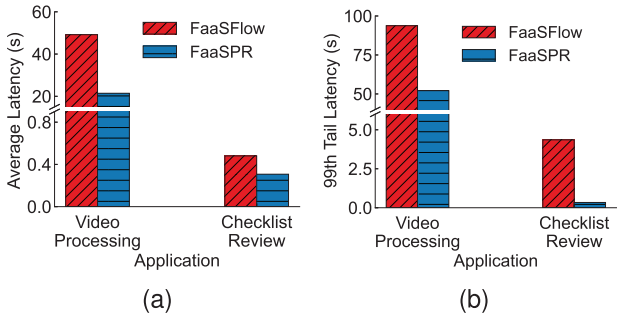


Fig. 15. Comparison between FaaSPR and FaaSFlow under 16 worker servers. (a) Average latency. (b) 99th tail latency.

55.00%, tail latency by 13.96%, 41.92%, 47.80%, and resource consumption by 1.58%, 0.67%, 12.32%, respectively. As each of the three simplified versions represents a technique in FaaSPR, the results indicate that all three techniques can reduce latency and resource consumption for FaaSPR.

As observed from the figure, DataTrans* performs similarly to FaaSFlow. This is because DataTrans* lacks the support of routing strategies, which can result in functions within the same workflow being routed to different horizontal groups, limiting the algorithm advantage. Additionally, Routing* shows a significant reduction in latency across all simplified versions. This is because the routing algorithm identifies potential data dependencies between randomly placed instances and reduces cross-server data transmission to some extent. Moreover, it balances instance loads and reduces function request queuing, further lowering processing latency.

E. Scalability

Finally, we evaluate the scalability of FaaSPR. Due to testbed limitations, this section includes both real and simulated experiments. We first demonstrate the performance of FaaSPR on a larger-scale cluster. Then we show the system overhead of the FaaSPR scheduler on up to 1024 nodes.

In the larger-scale cluster experiment, we increased the number of worker nodes to 16 and proportionally scaled the workflow invocation frequency, while keeping other conditions constant. We selected video processing and checklist review as representative second-level and millisecond-level workflows, respectively. Figure 15 shows the average and tail latencies for both workflows. Compared to FaaSFlow, FaaSPR reduces the average and tail latencies by 36.05%∼56.41% and 44.44%∼92.23%, respectively. FaaSPR reduces resource

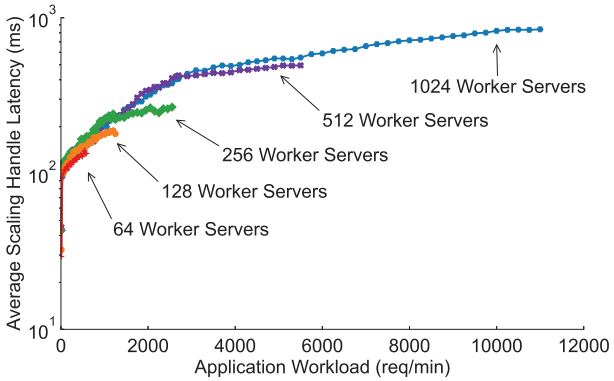


Fig. 16. The average latency of FaaSPR in handling scaling request.

usage for the video processing and checklist review workflows by 35.41% and 3.04%, respectively. This result omitted in the figure for space. These results are consistent with those from a smaller cluster, confirming that FaaSPR maintains stable performance at larger scales.

We then evaluate the system overhead of the FaaSPR scheduler. Figure 16 shows the average latency of the FaaSPR scheduler in handling scaling requests under different cluster sizes and invocation frequencies. We simulate the load of workers from 64 servers to 1024 servers on the scheduler. To eliminate burst traffic interference, we manually generate invocations at a stable frequency, instead of using the trace from the Azure Function Dataset [18]. The results show that, even with 1024 worker nodes and 11,000 req/min invocations, the handle latency remains below 900 milliseconds, much smaller than the minute-level scaling intervals of existing serverless platforms [26], [27]. This indicates that the scheduler can promptly respond to scaling requests, even in large-scale clusters, highlighting its scalability.

It is important to note that the scaling handle latency is unrelated to the workflow scheduling overhead. Scaling handle latency is the duration the scheduler takes to update placement and routing strategies for scaling requests. When processing user workflow requests, the scheduler applies the weighted random algorithm to assign function requests to worker servers. Internally, the generation of placement and routing strategies occurs in parallel with workflow request scheduling. The scheduler updates the random weights only after generating new routing strategies. In our experiments, we observed that the workflow scheduling overhead remains consistently at the microsecond level, which is negligible for serverless workflow latency.

VIII. Related Work

In this section, we review the related work of FaaSPR from the perspectives of serverless workflow processing and placement and routing strategies.

Serverless Workflow Optimization. Serverless workflow optimization is not an emerging research topic. Numerous studies have aimed to improve workflow execution speed or enhance deployment cost-effectiveness from various perspectives, such as instance cold starts [18], [19], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55],

[56], [57], [58], [59], intermediate data transmission [12], [13], [16], [17], [60], task scheduling [61], [62], platform architecture [17], [60], [63], and so on.

In terms of cold starts, early works relied on heuristic algorithms to reserve instances according to function invocation frequencies [18], [42]. This method is simple and effective but overlooks the dependencies among functions within workflows. By incorporating workflow structure, ORION [43], Kraken [59], and Xanadu [44] propose various heuristic traffic prediction methods, while AQUATOPE [45] predicts user traffic through neural networks. Accurate traffic prediction allows instance scaling requests to match actual user traffic better, enhancing the optimization potential of FaaSPR. Beyond traffic prediction, numerous studies have focused on reducing cold start overhead by accelerating image transmission times [19], [46], [47], [48], [49], [50], speeding up instance initialization [51], [52], [53], [54], [55], [57], [58], or improving instance reusability [56]. These works are orthogonal to FaaSPR and can further enhance it.

Regarding data transmission, due to the high overhead of remote storage services, most works replace remote storage services with local databases [12], [13], [16], [17], [60]. FaaSFlow [17] and SPRIGHT [60] utilize shared memory to handle data exchange between instances deployed on the same server. SAND [16] adopts a similar approach but deploys the workflow within a single container to reduce cold start overhead. When remote data exchange is unavoidable, SONIC [12] and Pocket [13] select appropriate storage solutions based on user requirements to optimize transmission performance and deployment costs. FaaSPR, consistent with previous work, utilizes shared memory to reduce data transmission overhead. Additionally, some approaches optimize serverless workflow processing through task scheduling [61], [62] and control plane accelerating [17], [60], [63], which are also orthogonal to FaaSPR and can serve as a complement.

There is a lack of efficient placement and routing optimization for serverless workflow processing. Palette [20] deploys instances within the same workflow together to reduce data exchange overhead. However, the placement strategy of Palette relies entirely on user annotations and cannot distinguish between different functions within a workflow. FaaSFlow [17] groups and places instances based on the workflow structure and function dependencies, but the strategy does not consider server hardware resource limitations, which restricts its performance. In contrast, FaaSPR overcomes the grouping limitations imposed by server resource capacity while eliminating various overheads such as cold starts and function request queuing, effectively optimizing workflow execution performance.

Placement and Routing. placement and routing optimization has undergone extensive and in-depth research in integrated circuits [64], [65], [66], [67], virtual network functions [68], [69], [70], [71], [72], and edge computing [68], [73], [74], [75], [76], [77] to determine the deployment locations of functional units and the routing paths for user requests or electronic signals. However, the placement and routing optimization in serverless workflow processing have different optimization goals and constraints, making it difficult to apply previous approaches directly. For instance, the integrated circuit domain focuses on the spatial arrangement of circuits [64], [65], [66], [67], while virtual function networks and edge computing emphasize the spatial relationship between function units, the core network, and users [68], [69], [70], [71], [72], [73], [74], [75], [76], [77]. Xu et al. proposed a placement and routing strategy for deploying serverless workflows in edge networks [21]. However, their model relies on an edge scenario, where there is no remote storage service (data transmission is achieved through limited network topology between servers) and does not consider bandwidth constraints, making it unsuitable for serverless platforms in data centers. Linear programming is a commonly used heuristic algorithm in the above fields [66], [69], [75]. FaaSPR adopts the concept of linear programming and designs a targeted placement and routing optimization based on the characteristics of serverless workflow processing.

IX. Conclusion

This paper presents FaaSPR, which optimizes instance placement and request routing for serverless workflow processing. On the one hand, FaaSPR eliminates inter-server data transmission through horizontal and vertical instance grouping and reduces cold start overhead by heuristic instance reusing strategy. On the other hand, the multi-stage LP-based routing algorithm of FaaSPR cooperates with the placement strategy to avoid queuing delays for function requests while maximizing local data transmission. As shown in experiments, FaaSPR effectively reduces the average and tail response latencies of serverless workflows across various evaluation scenarios while reducing user resource usage and deployment costs. We hope our work opens the door to more placement and routing optimizations for serverless workflow processing.

References

[1] AWS Lambda.(2024). *Understandig Virtualization*. [Online]. Available: https://aws.amazon.com/lambda/
[2] Microsoft.(2024). *Microsoft Azure Functions*. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/
[3] Google.(2024). *Google Cloud Functions*. [Online]. Available: https://cloud.google.com/functions/
[4] Alibaba.(2024). *Build an Elastic and Highly Available Audio and Video Processing System in a Serverless Architecture*. [Online]. Available: https://www.alibabacloud.com/help/en/fc/use-cases/build-an-elastic-and-highly-available-audio-and-video-processing-system-in-a-serverless-architecture
[5] Azure.(2019). *Serverless Microservices Reference Architecture—Architecture Overview*. [Online]. Available: https://github.com/azure-samples/serverless-microservices-reference-architecture/blob/main/documentation/architecture-overview.md
[6] AWS.(2024). *AWS Step Functions Use Cases*. [Online]. Available: https://aws.amazon.com/step-functions/use-cases/?nc1=h_ls
[7] AWS.(2024). *AWS Step Functions: Visual Workflows for Distributed Applications*. [Online]. Available: https://aws.amazon.com/step-functions
[8] Azure.(2024). *Overview: Azure Logic Apps*. [Online]. Available: https://learn.microsoft.com/en-us/azure/logic-apps/logic-apps-overview
[9] Google.(2024). *Workflows—Google Cloud*. [Online]. Available: https://cloud.google.com/workflows
[10] Apache.(2023). *OpenWhisk Composer*. [Online]. Available: https://github.com/apache/openwhisk-composer
[11] M. Swiderski. (2024). *Workflow as a Function Flow With Automatiko*. [Online]. Available: https://knative.dev/blog/articles/workflow-as-function-flow/

[12] A. Mahgoub et al., "SONIC: Application-aware data passing for chained serverless applications," in *Proc. USENIX Annu. Tech. Conf.*, Santa Clara, CA, USA, Jan. 2021, pp. 285–301.

[13] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proc. 16th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Boston, MA, USA, Feb. 2019, pp. 193–206.

[14] AWS.(2024). *Cloud Object Storage—Amazon S3*. [Online]. Available: https://aws.amazon.com/s3/

[15] Azure.(2024). *Azure Blob Storage*. [Online]. Available: https://azure.microsoft.com/en-us/services/storage/blobs/

[16] I. E. Akkus et al., "SAND: Towards high-performance serverless computing," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA, Jul. 2018, pp. 923–935.

[17] Z. Li et al., "FaaSFlow: Enable efficient workflow execution for function-as-a-service," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Lausanne, Switzerland, Feb. 2022, pp. 782–796.

[18] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, Virtual Event, Jul. 2020, pp. 205–218.

[19] A. Wang et al., "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute," in *Proc. USENIX Annu. Tech. Conf.*, Santa Clara, CA, USA, Jan. 2021, pp. 443–457.

[20] M. Abdi et al., "Palette load balancing: Locality hints for serverless functions," in *Proc. 18th Eur. Conf. Comput. Syst.*, Rome, Italy, May 2023, pp. 365–380.

[21] Z. Xu et al., "Stateful serverless application placement in MEC with function and state dependencies," *IEEE Trans. Comput.*, vol. 72, no. 9, pp. 2701–2716, Sep. 2023.

[22] E. Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," 2019, *arXiv:1902.03383*.

[23] AWS Lambda.(2024). *Memory and Computing Power*. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html

[24] S. Jaktholm. (2024). *Aws Network Benchmark*. [Online]. Available: https://github.com/sjakthol/aws-network-benchmark

[25] AWS.(2024). *Lambda Function Scaling*. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/scaling.html

[26] AWS Lambda.(2024). *Monitoring Concurrency*. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html\#managing-provisioned-concurency

[27] Knative.(2024). *Configuring Scale Bounds*. [Online]. Available: https://knative.dev/docs/serving/autoscaling/scale-bounds/

[28] AWS.(2024). *Scheduled Scaling for Application Auto Scaling*. [Online]. Available: https://docs.aws.amazon.com/autoscaling/application/userguide/application-auto-scaling-scheduled-scaling.html

[29] Knative.(2024). *Configuring Concurrency*. [Online]. Available: https://knative.dev/docs/serving/autoscaling/concurrency/

[30] AWS.(2024). *Placement Groups*. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html

[31] Z. Rafalovich. (2019). *Introducing Proximity Placement Groups*. [Online]. Available: https://azure.microsoft.com/en-us/blog/introducing-proximity-placement-groups/

[32] Google.(2024). *About Placement Policies*. [Online]. Available: https://cloud.google.com/compute/docs/instances/placement-policies-overview

[33] AWS.(2024). *Configure Lambda Function Memory*. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html

[34] Google.(2024). *Configure Memory Limits*. [Online]. Available: https://cloud.google.com/run/docs/configuring/services/memory-limits

[35] Kubernetes.(2024). *Open-Source System for Automating Deployment, Scaling, and Management of Containerized Applications*. [Online]. Available: https://kubernetes.io

[36] Redis.(2024). *Data Structure Store*. [Online]. Available: https://redis.io/

[37] SciPy.(2024). *Fundamental Algorithms for Scientific Computing in Python*. [Online]. Available: https://scipy.org

[38] FFmpeg.(2024). *A Complete, Cross-Platform Solution to Record, Convert and Stream Audio and Video*. [Online]. Available: https://ffmpeg.org

[39] AWS Lambda.(2024). *Create a Serverless Workflow With AWS Step Functions and AWS Lambda*. [Online]. Available: https://aws.amazon.com/cn/tutorials/create-a-serverless-workflow-step-functions-lambda/

[40] Amazon.(2024). *Quickstart: Create a Python Durable Functions App*. [Online]. Available: https://aws.amazon.com/cn/tutorials/create-a-serverless-workflow-step-functions-lambda/

[41] Google.(2024). *Workflows Tutorial*. [Online]. Available: https://cloud.google.com/functions/docs/tutorials/workflows/

[42] R. B. Roy, T. Patel, and D. Tiwari, "IceBreaker: Warming serverless functions better with heterogeneity," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Lausanne, Switzerland, Feb. 2022, pp. 753–767.

[43] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs," in *Proc. 16th USENIX Symp. Operating Syst. Design Implement.*, Carlsbad, CA, USA, 2022, pp. 303–320.

[44] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proc. 21st Int. Middleware Conf.*, Delft, The Netherlands, 2020, pp. 356–370.

[45] Z. Zhou, Y. Zhang, and C. Delimitrou, "AQUATOPE: QoS-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, Vancouver, BC, Canada, Mar. 2023, pp. 1–14.

[46] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy Docker containers," in *Proc. USENIX Conf. File Storage Technol.*, Santa Clara, CA, USA, Feb. 2016, pp. 181–195.

[47] C. Zheng et al., "Wharf: Sharing Docker images in a distributed file system," in *Proc. ACM Symp. Cloud Comput.*, Carlsbad, CA, USA, Oct. 2018, pp. 174–185.

[48] H. Liu et al., "CFS: A distributed file system for large scale container platforms," in *Proc. Int. Conf. Manage. Data*, Amsterdam, The Netherlands, Jun. 2019, pp. 1729–1742.

[49] H. Li, Y. Yuan, R. Du, K. Ma, L. Liu, and W. Hsu, "DADI: Block-level image service for agile and elastic application deployment," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, Virtual Event, Jul. 2020, pp. 727–740.

[50] Z. Zhang et al., "VMThunder: Fast provisioning of large-scale virtual machine clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 12, pp. 3328–3338, Dec. 2014.

[51] D. Du et al., "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Lausanne, Switzerland, Mar. 2020, pp. 467–481.

[52] E. Oakes et al., "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA, 2018, pp. 57–70.

[53] H. A. Lagar-Cavilla et al., "SnowFlock: Rapid virtual machine cloning for cloud computing," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, Nuremberg, Germany, Apr. 2009, pp. 1–12.

[54] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2021, pp. 559–572.

[55] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "SEUSS: Skip redundant paths to make serverless fast," in *Proc. 15th Eur. Conf. Comput. Syst.*, Heraklion, Greece, Apr. 2020, pp. 32:1–32:15.

[56] K.-T.-A. Wang, R. Ho, and P. Wu, "Replayable execution optimized for page sharing for a managed runtime environment," in *Proc. 14th EuroSys Conf.*, Dresden, Germany, Mar. 2019, pp. 39:1–39:16.

[57] W. Shin, W.-H. Kim, and C. Min, "Fireworks: A fast, efficient, and safe serverless framework using VM-level post-JIT snapshot," in *Proc. 17th Eur. Conf. Comput. Syst.*, Rennes, France, Mar. 2022, pp. 663–677.

[58] Z. Li et al., "Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing," in *Proc. USENIX Annu. Tech. Conf.*, Carlsbad, CA, USA, 2022, pp. 69–84.

[59] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic DAGs in serverless platforms," in *Proc. ACM Symp. Cloud Comput.*, Seattle, WA, USA, 2021, pp. 153–167.

[60] S. Qi, L. Monis, Z. Zeng, I.-C. Wang, and K. K. Ramakrishnan, "SPRIGHT: Extracting the server from serverless computing! High-performance eBPF-based event-driven, shared-memory processing," in *Proc. ACM SIGCOMM Conf.*, Amsterdam, The Netherlands, Aug. 2022, pp. 780–794.

[61] C. Jin et al., "Ditto: Efficient serverless analytics with elastic parallelism," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, Sep. 2023, pp. 406–419.

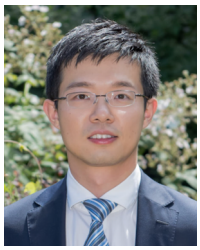
[62] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: NIMBLE task scheduling for serverless analytics," in *Proc. USENIX Symp. Networked Syst. Design Implement.*, Boston, MA, USA, Jan. 2021, pp. 653–669.
[63] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proc. 11th ACM Symp. Cloud Comput.*, Oct. 2020, pp. 1–15.
[64] C. Ababei et al., "Placement and routing in 3D integrated circuits," *IEEE Design Test Comput.*, vol. 22, no. 6, pp. 520–531, Jun. 2005.
[65] R. Cheng and J. Yan, "On joint learning for solving placement and routing in chip design," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, Jan. 2021, pp. 16508–16519.
[66] M. Canesche et al., "TRAVERSAL: A fast and adaptive graph-based placement and routing for CGRAs," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 40, no. 8, pp. 1600–1612, Aug. 2021.
[67] W.-H. Chen and Y.-W. Chang, "Graph-based simultaneous placement and routing for two-dimensional directed self-assembly technology," in *Proc. 60th ACM/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Jul. 2023, pp. 1–6.
[68] S. Yang, F. Li, S. Trajanovski, X. Chen, Y. Wang, and X. Fu, "Delay-aware virtual network function placement and routing in edge clouds," *IEEE Trans. Mobile Comput.*, vol. 20, no. 2, pp. 445–459, Feb. 2021.
[69] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in *Proc. IEEE 4th Int. Conf. Cloud Netw. (CloudNet)*, Niagara Falls, ON, Canada, Oct. 2015, pp. 171–177.
[70] N. He et al., "Leveraging deep reinforcement learning with attention mechanism for virtual network function placement and routing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 4, pp. 1186–1201, Apr. 2023.
[71] T. Gao et al., "Cost-efficient VNF placement and scheduling in public cloud networks," *IEEE Trans. Commun.*, vol. 68, no. 8, pp. 4946–4959, Aug. 2020.
[72] S. Yuan, Y. Sun, and M. Peng, "Joint network function placement and routing optimization in dynamic software-defined satellite-terrestrial integrated networks," *IEEE Trans. Wireless Commun.*, vol. 23, no. 5, pp. 5172–5186, May 2024.
[73] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas, "Service placement and request routing in MEC networks with storage, computation, and communication constraints," *IEEE/ACM Trans. Netw.*, vol. 28, no. 3, pp. 1047–1060, Jun. 2020.
[74] Z. Gao et al., "Deep reinforcement learning-based policy for baseband function placement and routing of RAN in 5G and beyond," *J. Lightw. Technol.*, vol. 40, no. 2, pp. 470–480, Jan. 15, 2022.
[75] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas, "Joint service placement and request routing in multi-cell mobile edge computing networks," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Paris, France, Apr. 2019, pp. 10–18.
[76] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano, "NEPTUNE: A comprehensive framework for managing serverless functions at the edge," *ACM Trans. Auto. Adapt. Syst.*, vol. 19, no. 1, pp. 1–32, Mar. 2024.
[77] Y. Hu, H. Wang, L. Wang, M. Hu, K. Peng, and B. Veeravalli, "Joint deployment and request routing for microservice call graphs in data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 11, pp. 2994–3011, Nov. 2023.

Yunshan Jia received the B.S. degree from Peking University, Beijing, China, in 2021, where she is currently pursuing the Ph.D. degree with the School of Computer Science. She is working on serverless workflow orchestration. Her research interests include cloud computing and wireless communication.

Chao Jin received the B.S. degree from Peking University, Beijing, China, in 2023, where he is currently pursuing the Ph.D. degree with the School of Computer Science. He is working on serverless workflow orchestration. His research interests include cloud computing and machine learning systems.

Qing Li was a Post-Doctoral Researcher at Peking University from 2022 to 2024. She is currently an Associate Professor with the School of Computer Science (National Pilot Software Engineering School), Beijing University of Posts and Telecommunications. Her main research interests include satellite computing system optimization and resource scheduling theory in edge and cloud computing. She received the Best Paper Award from IEEE EDGE 2021 and the Distinguished Paper Award from ICSOC 2023.

Xuanzhe Liu (Senior Member, IEEE) is currently a Full Professor with the School of Computer Science, Peking University, Beijing, China. Most of his recent efforts have been published at prestigious conferences, such as WWW, ICSE, FSE, SOSP, SIGCOMM, NSDI, MobiCom, and MobiSys, and journals, such as *ACM Transactions on Software Engineering and Methodology*/*ACM Transactions on Information Systems* and IEEE TRANSACTIONS ON SOFTWARE ENGINEERING/IEEE TRANSACTIONS ON MOBILE COMPUTING/IEEE TRANSACTIONS ON SERVICES COMPUTING. His research interests include service-based software engineering and systems. He is a Distinguished Member of ACM and CCF. For more information: http://www.liuxuanzhe.com/.



Xin Jin (Senior Member, IEEE) received the Ph.D. degree from Princeton University in 2016. He is currently an Associate Professor (with Tenure) with the School of Computer Science, Peking University. His research interests include computer systems, networking, and cloud computing.