StepConf: SLO-Aware Dynamic Resource Configuration for Serverless Function Workflows

Zhaojie Wen, Yishuo Wang, Fangming Liu*
National Engineering Research Center for Big Data Technology and System,
Key Laboratory of Services Computing Technology and System, Ministry of Education,
School of Computer Science and Technology, Huazhong University of Science and Technology, China

***Abstract*—Function-as-a-Service (FaaS) offers a fine-grained resource provision model, enabling developers to build highly elastic cloud applications. User requests are handled by a series of serverless functions step by step, which forms a function-based workflow. The developers are required to set proper resource configuration for functions, so as to meet service level objectives (SLOs) and save cost. However, developing the resource configuration strategy is challenging. It is mainly because execution of cloud functions often suffers from cold start and performance fluctuation, which requires a dynamic configuration strategy to guarantee the SLOs. In this paper, we present StepConf, a framework that automates the resource configuration for functions as the workflow runs. StepConf optimizes memory size for each function step in the workflow and takes inter and intra-function parallelism into consideration. We evaluate StepConf on AWS Lambda. Compared with baselines, the experimental results show that StepConf can save cost up to 40.9% while ensuring the SLOs.**

***Index Terms*—serverless computing, configuration optimization, resource provisioning**

I. Introduction

Function-as-a-Service (FaaS) is a new serverless computing paradigm that enables developers to run code on the cloud without maintaining and operating the cloud resources [1]–[4]. Developers only need to submit function codes to FaaS platforms, where computing resources are provisioned and functions are executed seamlessly. Hence, the developers only need to focus on the business logic, which accelerates the progress of application development and saves operational cost. With FaaS platforms like AWS Lambda [5], developers are able to use hundreds of CPU cores in a short period of time by invoking massive functions concurrently.

Benefiting from the fine-grained high elasticity of FaaS, a lot of applications were built on top of serverless functions, including video processing [6], [7], machine learning [8]–[10], code compilation [11], big-data analytic [12], [13], etc. To migrate applications to FaaS platforms, developers need to decouple monolithic application into multiple functions, which generates complex function-base workflows. However, it is challenging for developers to optimize cost of functions and guarantee performance of workflows. The main reasons are as follows:

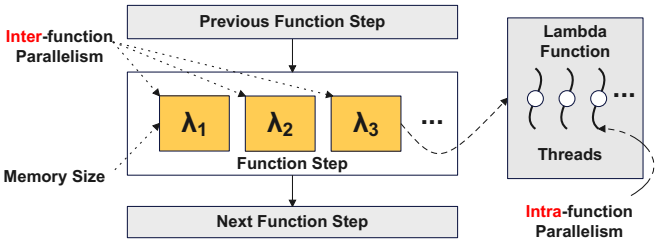

Fig. 1. The architecture of inter and intra-function parallelism.

Vague impact of resource configuration: The cost and performance of functions highly depends on resource parameters configured by FaaS users. However, it is unclear that how to select proper resource parameters can achieve high performance and low cost [14]–[16]. Hence, it is challenging for FaaS users to determine the resource configuration of functions.

Exponential growth of configuration space: FaaS users are required to configure a series of resource parameters for functions. As the number of functions in workflow grows, the decision space for resource parameters increases exponentially, which makes it challenging to obtain the optimal configuration. Meanwhile, dynamically adjusting the configuration according to different SLOs further complicates the resource configuration problem.

For many FaaS platforms, we are only required to configure the memory size of the function, which directly affects the resources allocated by the function. Besides function memory size, what other configurations will affect the performance of the function workflow? For many computing tasks, parallelism is the key to improving workflow performance. As shown in Fig. 1, serverless developers could either take advantage of multi-core computing resources of a single function by coding with multi-threads and processes (intra-function parallelism) or mapping the tasks to concurrent functions in parallel (inter-function parallelism).

Through the experiments conducted on AWS, we find that inter and intra-function parallelism are crucial factors that we need to consider, in addition to function memory size.

*The corresponding author is Fangming Liu (fangminghk@gmail.com).

This work is supported by Huawei, by National Key Research & Development (R&D) Plan under grant 2017YFB1001703, by NSFC under grant 61722206 and 61761136014, and by National Program for Support of Top-notch Young Professionals in National Program for Special Support of Eminent Professionals.

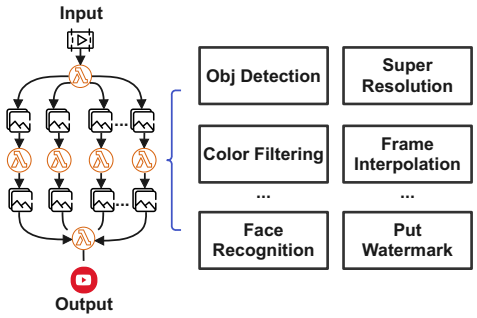

Fig. 2. Video processing workflow with different serverless functions.

Therefore, dealing with trade-offs between memory size and function parallelism is the key to optimizing workflow configurations. Further, we find that the performance fluctuation of functions is mainly caused by function cold start and performance variation of external storage services. Workflow SLO is often not guaranteed when an unexpected performance degradation occurs. For the above reasons, the developers often chose poor configurations for workflows, resulting in wasting costs and failing to meet SLOs.

In this paper, based on our insights, we present StepConf, an SLO-Aware dynamic cost optimization framework for serverless function workflows. Our main idea is to make configuration decisions dynamically in real-time before each function step is executed in the workflow and jointly optimize the inter and intra-function parallelism with function memory size. In this way, the running progress of the workflow can be corrected in real-time, and we can alleviate the influence of performance variation and guarantee the request SLOs while saving the cost.

Specifically, we first analyze the service mechanism of function workflow through experimental measurements on AWS Step Function. We analyze the experimental results to determine the best intra-function parallelism for each memory configuration. We also develop a piece-wise fitting model that achieves high performance prediction accuracy under different configurations, including memory sizes and inter and intra-function parallelism.

Secondly, StepConf transforms the overall workflow configuration optimization problem into a configuration decision problem for each function step. By optimizing the workflow configuration step by step and considering a trade-off between inter-function parallelism and function memory size, StepConf can guarantee the end-to-end performance SLOs and save more cost compared with baselines.

Finally, StepConf uses a heuristic optimization algorithm to deal with the configuration optimization problem which is proved to be NP-hard, offering suitable configurations for function steps while bringing little decision-making overhead.

To sum up, the main contributions of this paper are as follows:

- We present an end-to-end workflow performance and cost model with high prediction accuracy by piece-wise fitting the function performance under different configurations and function characteristics.
- We consider the inter and intra-function parallelism when formulating the dynamic optimization problem as an NP-hard problem and propose an efficient heuristic solution to solve it.
- We develop StepConf, a prototype of configuration optimization framework for AWS Step Function, which can guarantee workflow's SLO while saving the costs.

II. Background And Motivation

A. Application Architecture: Function Based Workflow

Typically, workflows are developed as massive functions of FaaS. A typical example of video processing workflows is shown in Fig. 2. We then discuss how critical resource parameters of functions impact performance and cost.

B. What Can We Configure?

1) Memory size: To explore the impact of increasing the memory size of the function on the performance of different programs, we deploy a function to determine whether each large number in a list is a prime number on AWS Lambda. We write both single-thread and multi-core-friendly versions of it. We set the multi-core ratio as the ratio of tasks that can fully utilize the multi-core resources. The higher the multi-core ratio is, the more multi-core friendly the program would be. We measure function performances under different multi-core ratios and memory sizes. According to the results shown in Fig. 3, we find that for a single-thread optimized program, the performance will no longer improve when the memory is increased to a certain value. The duration time of a multi-core optimized program is reduced proportionally as the memory increases. We confirm that the single-thread performance of the Lambda function would not be improved when it reaches 1769 MB. After more than 1769 MB, increasing memory can only bring more multi-core performance.

2) Inter-function parallelism: Multi-core optimization of user code is not easy. Some programs were originally not optimized for multi-core environments. Moreover, even for those multi-core-friendly programs, the computing resource of a single function is limited. So how to further improve performance? We can use the parallelism between different function instances to complete our computing tasks. For example, for a video processing workflow, we can invoke different functions concurrently to complete the processing tasks of different video frames respectively, as shown in Fig. 2. Although the computing power of a single function is limited, with a large number of inter-function parallelism, we can achieve outstanding performance.

Does increasing the degree of inter-function parallelism always bring benefits? The answer is not true. We evaluate the performance of Step Function workflows on AWS. From the results shown in Fig. 4, we find that the function step's mapping delay is relatively low when the degree of inter-function parallelism is smaller than about 30. However, the mapping delay is growing fast over 40. As shown in Fig. 5,

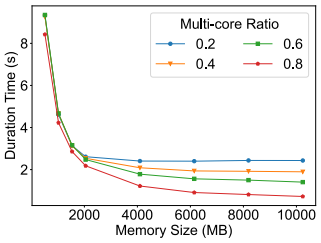

Fig. 3. Duration time of a function with different memory size and multi-core ratio.

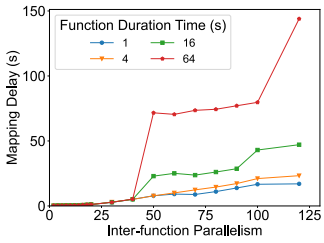

Fig. 4. Mapping delay of a function step with different function duration time and inter-function parallelism.

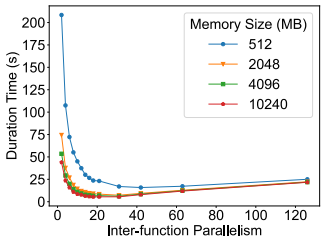

Fig. 5. Duration time of a function step with different memory size and inter-function parallelism.

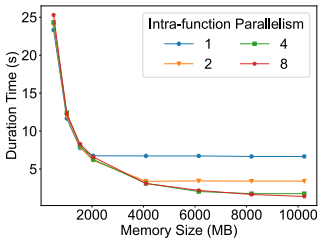

Fig. 6. Duration time of a function step with different memory size and intra-function parallelism.

we find that keeping increasing the degree of inter-function parallelism will not further accelerate the computation. This is because the mapping overheads become a bottleneck. As a result, we need to optimize the selection of function memory size and degree of inter-function parallelism to achieve cost-effective.

3) Intra-function parallelism: For the programs with poor multi-core optimization, the strategy of intra-function parallelism will undoubtedly improve the resource utilization of the function and improve the running efficiency. As shown in the Fig 6, we find that AWS allocates different numbers of available vCPUs for the functions in different memory sizes. Obviously, choosing the appropriate intra-function parallelism can improve the performance of the workflow and reduce costs at the same time.

C. Per-step Dynamic Configuration for SLO Guarantee

Function cold starts can be defined as the set-up time required to get a serverless application's environment up and running when it is invoked for the first time within a specified period of time. Although cloud vendors have tried their best to reduce the function cold start overhead, they still bring a second-level cold start delay when starting some relatively large functions [17], [18]. An unexpected cold start in the function workflow will cause degradation in the performance. In addition, other cloud services (such as storage [19], database [20], etc.) are often served to serverless functions, and these external services will bring inevitable performance fluctuations. This volatility will make the duration time of the function unpredictable, making it more difficult to optimize the whole workflow's end-to-end performance and guarantee user SLOs. All in all, it is necessary for us to adjust the configuration of the workflow in real-time dynamically.

III. Model Design and Problem Formulation

A. Workflow Execution Model

Serverless function workflow contains different function steps to finish the logic of an application. We first provide the definition of function step in the Definition 1.

Definition 1. *The process of performing parallel tasks by concurrently running the instances of the same serverless function is called a function step. When the concurrency is 1, a single function is also called a special case of function step.*

We use a directed acyclic graph $G = (V, E)$ to characterize the function workflow. The vertices $V = \{v_1, v_1, ..., v_n\}$ represent the n function steps in the workflow. The vertice with 0 in-degree is the starting point of the workflow, i.e., the first function step to be executed. The vertice with 0 out-degree is the end of the workflow, i.e., the last function step which is executed. The edges $E = \{\widehat{v_i v_j} \mid 1 \leq i \neq j \leq N\}$ represent the dependency of function steps, where $\widehat{v_i v_j}$ represents the edge $v_i \rightarrow v_j$, i.e., the operation of function step v_j depends on v_i.

For a function step v_i, it will start to run if and only if all its dependencies are completed. Let the path $L \in \mathcal{L}$ denote the collection of vertices that travel from the starting point to the end. We define the running time of function step v_i as t_i, and define the completion time of the function workflow as T. It is easy to know that the completion time T depends on the path with highest running time, which is formulated as $T = \max_{L \in \mathcal{L}} \sum_{v_i \in L} t_i$.

We finally define the cost of function step v_i as c_i, then the total cost of processing the workflow can be calculated by $C = \sum_{v_i \in V} c_i$.

B. Function Step Model

After providing the workflow execution model, we further model the duration time and cost of each function step in this section.

1) Mapping overhead: For each function step v_i, we define the memory size, the degree of inter-function parallelism, and the degree of intra-function parallelism as m_i, p_i, and q_i, respectively.

The running process of a function step is shown in the Fig. 7. When the workflow enters the function step v_i, p_i function instances will be invoked concurrently according to the inter-function parallelism. It is worth noting that functions will not be started simultaneously due to the orchestration overheads and cold start. In fact, these function instances are started in sequence, which causes the mapping delay.

As shown in Fig. 7, we define the mapping delay as the time duration from the beginning of the function step until the sandbox of the function instance is prepared by the cloud vendor. We denote the mapping delay in the function step v_i as $t_i^{map}(p)$. We consider function cold start time is included in the mapping delay.

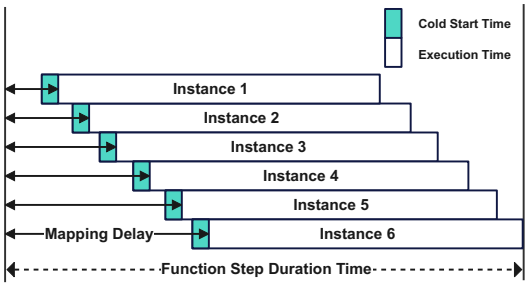

Fig. 7. Function instances with inter-function parallelism in a function step. The function instances are scheduled by the cloud vendors with different mapping delays.

2) Function Step Duration Time: As previous discussed, each function step contains function instances to run in parallel. We denote the jth function instance in function step v_i as $v_{i,j}$ and its duration time as $t_{i,j}$. The relationship between function step duration time and the duration time of instances is as $t_i = \max_{1 \le j \le p_i} [t_i^{map}(j) + t_{i,j}]$.

In order to simplify the expression, we ignore the performance fluctuation of functions computing time. We assume that the last invoked function in the function step is the last to finish. So, we denote the average compute duration time of the function as $\hat{t_i}$, then we have $t_i = t_i^{map}(p_i) + \hat{t_i}$.

Intuitively, the duration time of a function is related to the number of allocated jobs and memory size. We denote the total number of jobs in a function step as γ_i. According to the inter-function parallelism p_i, the number of jobs for a function instance is $\overline{\gamma}_i = \frac{\gamma_i}{p_i}$.

From the experiment results in Fig. 3, the duration time of function is inversely proportional to memory size, which means the multi-core performance of the function increases almost linearly with the memory size. The performance of function is also affected by the intra-function parallelism. We set the memory size that the cloud vendor exactly allocates a full single-core resource for a function as m_s, where s stands for single and its relative performance is denoted as δ_s.

We use $\min(\frac{m_i}{m_s}, q_i)$ to indicate the upper bound of multi-core speed up ratio for single-thread-optimized function with intra-function parallelism. However, for programs that have adopted multi-core optimization, the effect of intra-function parallelism will be discounted. In this situation, the relationship is no longer inversely proportional, so we fit it with a exponential function to improve accuracy for prediction.

From this, we develop a piece-wise fitting method. For the programs that are multi-core-friendly (case 1), we use an exponential function for fitting; For programs that can only take advantage of single-thread performance or the memory size is lower than m_s (case 2), we use an inverse proportional function for fitting:

$$\hat{t_i} = \begin{cases} (\alpha_{i,1} \cdot \overline{\gamma}_i + \beta_{i,1}) e^{-\alpha_{i,2} \cdot \min(\frac{m_i}{m_s}, q_i)} + \phi_{i,1}, & \text{for case 1} \\ \frac{\alpha_{i,2} \cdot \overline{\gamma}_i}{\delta_s \cdot \min(\frac{m_i}{m_s}, q_i) + \beta_{i,2}} + \phi_{i,2}, & \text{for case 2} \end{cases} \quad (1)$$

where $\alpha_{i,1}, \alpha_{i,2}, \beta_{i,1}, \beta_{i,2}, \phi_{i,1}, \phi_{i,2}$ are the model parameters.

TABLE I
MAIN NOTATIONS

Notation	Definition
G	Function workflow DAG
v_i	The ith function step in the workflow
$\mathcal{L}_{critical}$	The critical path of a graph
T	Total duration time of the function workflow
C	Total cost of the function workflow
$\mathcal{L}$	The set of all possible path in a graph
G_i	The sub DAG starting from function step v_i
$t_{i,j}$	The duration time of jth function instance with inter-function parallelism in the function step v_i
m_i	The memory size configuration of function step
p_i	The degree of inter-function parallelism of function step v_i
q_i	The degree of intra-function parallelism of function step v_i
t_i	The total duration time of function step v_i
c_i	The total cost of function step v_i
Θ_i	All configuration combinations of function step v_i
γ_i	The number of jobs of function step v_i
$x_i(\theta)$	A binary variable which indicates whether v_i is configured to θ
$t_i(\theta)$	The total duration time of v_i configured to θ
$c_i(\theta)$	The total cost of v_i configured to θ
$\mathcal{S}$	The end to end SLO of function workflow
s_i	The sub-SLO of function step v_i
$\mathcal{T}_i^*$	The latest finish timestamp of G_i
τ_k^*	The latest finish timestamp of v_k

3) Cost of function step: Let c_i denote the cost of the function step v_i in the workflow. Different cloud vendors have similar pricing models for function workflow. In the following, we take the pricing model of AWS as a representation [21]. We denote the price for per GB-second of function as μ_0, and the price for function requests and orchestration as μ_1, where μ_0 and μ_1 are constants, then we have $c_i = p_i \cdot (t_i \cdot m_i \cdot \mu_0 + \mu_1)$.

C. SLO-Aware Configuration Optimization

For dynamic optimizing the configuration, we need to find out the relationship between different configurations of function steps and workflow's cost and performance. For clarity, the important notations are listed in Table I.

1) Configuration options: We denote the configuration of the function step v_i as $\theta_i = (m_i, p_i, q_i)$. It is worth noting that the memory size of all parallel functions of each function step is the same as the memory configuration of the function step.

Let $\mathcal{M}$ denote the set of memory configurations provided by the cloud platform, and $\mathcal{P}$ denote the optional inter-function parallelism set. From the measurement result in Fig. 6, we can obtain the appropriate intra-function parallelism for each memory sizes. Besides, implementing intra-function parallelism will use more memory and may exceed the function memory limit. So we use $\mathcal{Q}$ to denote the set for the best degree of intra-function parallelism in available. Then, we can define the set of configurations of function step v_i as:

$$\Theta_i = \{(m_i, p_i, q_i) \mid m_i \in \mathcal{M}, p_i \in \mathcal{P}, q_i \in \mathcal{Q}\}. \quad (2)$$

We use a binary variable $x_i(\theta)$ to indicate whether the function step v_i is configured to θ:

$$x_i(\theta) = \begin{cases} 0, & \text{function step } v_i \text{ is not configured to } \theta \\ 1, & \text{function step } v_i \text{ is configured to } \theta \end{cases}. \quad (3)$$

As each function can use only one configuration, we have:

$$\sum_{\theta \in \Theta_i} x_i(\theta) = 1, \forall v_i \in V. \tag{4}$$

2) Problem formulation: Our goal is to optimize the end-to-end performance of the workflow and minimize the cost while meeting the SLO of the workflow. Let $\mathcal{S}$ denote the SLO of the workflow request, and we have the SLO constraint:

$$T = \max_{L \in \mathcal{L}} \sum_{v_i \in L} \sum_{\theta \in \Theta_i} t_i(\theta) \cdot x_i(\theta) < \mathcal{S}. \tag{5}$$

Our **F**unction **W**orkflow **C**onfiguration **P**roblems (**FWCP**) is formulated by:

$$\min C = \min \sum_{v_i \in V} \sum_{\theta \in \Theta_i} c_i(\theta) \cdot x_i(\theta) \tag{6}$$
$$\text{subject to: } (2)(3)(4)(5).$$

IV. Solutions

A. Straightforward Algorithm for the Problem *FWCP*

Consider the SLO of the workflow request is $\mathcal{S}$, the start timestamp of the workflow is $\mathcal{T}_0$, and the timestamp before entering the function step v_i is τ_i. For the function step v_i, we define the graph consists of all paths starting from v_i to the destination as G_i. It is easy to find that G_i is the sub-graph of G, i.e., $G_i \subseteq G$.

Algorithm 1 presents an intuitive idea to choose the optimal configuration for each function step. To be specific, we can obtain the best configuration for each function step by calculating the sub-SLO of each function step (i.e., Line 5) and then solving (6) (i.e., Line 6). Lines 7-8 are used to search for the possible configuration to ensure the workflow SLO. Lines 9-11 show that if there is no feasible solution which means that the SLO cannot be guaranteed, and we let function step be configured with the best performance θ_{max}, trying to catch up the SLO.

Although Algorithm 1 is straightforward, the following Theorem 1 shows that the problem (6) is NP-hard, which makes this algorithm impractical.

Theorem 1. *Problem **FWCP** (6) is NP-hard.*

Proof. We construct a polynomial-time reduction to (6) from the multiple-choices knapsack problem (7)(8), a classic optimization problem which is known to be NP-hard.

$$\max \sum_{i=1}^{m} \sum_{j \in N_i} \mu_{ij} x_{ij} \tag{7}$$

$$\text{subject to}: \begin{cases} \sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} \leq C \\ x_{ij} \in \{0,1\}, 1 \leq i \leq m, j \in N_i \\ \sum_{j \in N_i} x_{ij} = 1, 1 \leq i \leq m \end{cases}. \tag{8}$$

Given an instance $A = (m, \mu_{ij}, w_{ij}, C, x_{ij})$ of the knapsack problem, we map it to an instance of the (6) with $A^{'} = (n \leftarrow m, -c_i(\theta) \leftarrow \mu_{ij}, t_i(\theta) \leftarrow w_{ij}, \mathcal{S} \leftarrow C, x_i(\theta) \leftarrow x_{ij})$. Clearly, the above mapping problem can be solved in polynomial time. Then, if there exists an algorithm that solves problem $A^{'}$, it solves the corresponding knapsack problem as well. As a result, multiple-choices knapsack problem can be treated as a special case of (6). Given the NP-hardness of the multiple-choices knapsack problem, (6) must be NP-hard as well. □

Algorithm 1: Dynamic Function Step Configuration Algorithm

Input:
The function workflow end to end SLO, $\mathcal{S}$;
Current function step v_i;
The function workflow start timestamp, $\mathcal{T}_0$;
Output:
Configuration for Function step v_i, θ_i;

```
1  Initialize;
2  Acquire the sub-graph G_i starting from v_i;
3  Acquire current timestamp, T_i;
4  while v_i is not the end do
5      Calculate the sub-SLO: S' ← S − (T_i − T_0);
6      Acquire {x_i(θ) | θ ∈ Θ_i} by solving (6);
7      if 1 ∈ x_i then
8          Find θ_i s.t. x_i(θ) = 1;
9      else No feasible solution:
10         θ_i ← θ_max;
11     return θ_i;
```

B. Problem Relaxation and Global-cached Most Cost-effective Critical Path Algorithm

In the following, we will introduce how to reduce the computation complexity through heuristic insights, and then propose a practical algorithm to configure each function step efficiently.

Note that (5) restricts the maximum duration of all paths $\mathcal{L}$ in the graph G by the request SLO. Therefore, we can set a sub-SLO s_i for each function step, where

$$\forall L \in \mathcal{L}, \sum_{v_i \in L} s_i \leq \mathcal{S}. \tag{9}$$

By doing this, we relax the constraint (5) by:

$$t_i = \sum_{\theta \in \Theta_i} t_i(\theta) \cdot x_i(\theta) < s_i. \tag{10}$$

Then we can reduce the problem (6) of optimizing a DAG to optimize the configuration of each function step v_i with sub-SLO constraints, which is formulated as follows:

$$\min c_i = \min \sum_{\theta \in \Theta_i} c_i(\theta) \cdot x_i(\theta) \tag{11}$$
$$\text{subject to: } (3)(10).$$

We next discuss how to set the sub-SLO for each function step. We denote the path with the longest duration time as the critical path [22]. And we regard the running time of each vertex (i.e., function step) in the critical path as its

weight and divide SLO among function steps proportionally based on it. However, it is hard to know the configuration of upcoming function steps and their running time in our online problem. Based on our insights, there is a configuration with the highest cost-effective ratio for each function step, defined as the ratio of performance to cost. We sort the configurations by the cost-effective ratio and find that near the configuration with the most cost-effective ratio, its cost-effective ratio is also relatively high. Based on this, we can have a better performance for optimization if we set the weights according to the duration time under the most cost-effective configuration.

In our problem, the performance is evaluated by multiplicative inverse of the duration time, i.e., t^{-1}, and hence the cost-effective ratio is $\frac{c}{t^{-1}}$. The best cost-effective configuration θ^* can be obtained by:

$$\theta^* = \arg\max_{\theta} \frac{c_i(\theta)}{t_i^{-1}(\theta)}. \tag{12}$$

We further denote $t_i(\theta^*)$ as the duration time of its function step with the configuration of most cost-effective ratio. And the sub-SLO for current function step v_i is given by:

$$s_i = \frac{t_i(\theta^*)}{\sum_{v_i \in \mathcal{L}_{critical}} t_i(\theta^*)} \cdot \mathcal{S}. \tag{13}$$

Our dynamic configuration strategy is essentially a distributed decision progress. There is no need for each configurator of function step to obtain the global information of the graph. We can establish a shared global cache for them to access the necessary data, which can reduce the amount of local computation. Based on this, we propose Global-cached Most Cost-effective Critical Path Algorithm in Algorithm 2.

To implement the Algorithm 2, we define a timestamp called the latest completion time τ_i^* for each function step v_i according to their SLOs. The latest completion time indicates the latest timestamp for the function step to guarantee SLO. It can be calculated by adding the timestamp of entering the current function step with its SLO.

Starting from configurator of a function step, we first get the current graph and the latest completion time of it from the global cache, and then calculate its sub-SLO (Lines 1-4). Next, we find the critical path of the graph according to the weight of each function step and assigns the sub-SLO for function steps (Lines 5-9). Then we can obtain the configuration of the current function step by solving (11) (Line 10).

For each sub-graph G^* connected on the critical path, we set its latest completion time to be the same with the drive-in node on the critical path (Lines 11-17). Then we save itself with its latest completion time to the global cache (Line 18). We also save the remains critical path and its latest completion time to the global cache (Lines 19-21). Finally, we return our configuration for current function step (Line 22).

Algorithm 2: Global-cached Most Cost-effective Critical Path Algorithm

Input: Current function step v_i
Output: The configuration θ_i for v_i

1 Acquire the current timestamp, $\mathcal{T}_{current}$ to initialize;
2 Acquire the sub-graph starting with v_i, G_i in global cache;
3 Acquire the latest finish timestamp for G_i, $\mathcal{T}_i^*$;
4 Calculate the sub-SLO of G_i by: $\mathcal{S}_i \leftarrow \mathcal{T}_i^* - \mathcal{T}_{current}$;
5 **foreach** *function step* $v_m \in V_i$ **do**
6 Acquire the configuration θ^* with the most cost-effective ratio with (12);
7 Set weight $t_m(\theta^*)$ for v_m;
8 Use weights to get the critical path $\mathcal{L}_i$ of G_i;
9 Calculate the SLO of v_i with (13);
10 Get the configuration θ_i for v_i by solving (11);
11 **foreach** *pair* v_j, v_k *where* $v_j, v_k \in \mathcal{L}_i$ **do**
12 **if** *exist sub-graph* G^* *between* v_j, v_k, *where* $V^* \cap \mathcal{L}_i = \varnothing$
13 **then**
14 Calculate the latest finish time of v_k, τ_k^*;
15 Calculate the sub-SLO of v_k, s_k;
16 Calculate the latest finish timestamp of G^* by:
17 $\mathcal{T}^* \leftarrow \tau_k^* - s_k$;
18 Store $(G^*, \mathcal{T}^*)$ in global cache;
19 $G^* \leftarrow \mathcal{L}_i \setminus \{v_i\}$;
20 $\mathcal{T}^* \leftarrow \mathcal{T}_i^*$;
21 Store $(G^*, \mathcal{T}^*)$ in global cache;
22 **return** the configuration θ_i;

Theorem 2. *The time complexity of Algorithm 2 is* $O(|V| + |E| + |\Theta|)$, *where* $|E|$ *is the number of edges in the current sub-graph,* $|V|$ *is the number of function steps in the the current sub-graph,* $|\Theta|$ *is the number of configuration combination choices.*

Proof. For Lines 1-4, the time complexity is at most $O(1)$. For Lines 5-7, the time complexity is at most $O(|V|)$. For Lines 8-9, the time complexity of getting the critical path and setting weights of a DAG is $O(|V| + |E|)$. For Line 10, the time complexity of solving (11) is at most $O(|\Theta|)$. And for Lines 11-22, the time complexity of getting the sub-graph and latest finish timestamp is at most $O(|V| + |E|)$. □

Theorem 2 shows the time complexity of our solution. It shows our solution can be solved in polynomial time. Some may argue that searching for the best configuration with (11) is hard when the configuration space is large. To deal with it, we can maintain a local record of optimal configuration and replace the sub-optimal configuration when there is a configuration choice with better performance at the same cost or the less cost with the same performance. Once established, we do not have to traverse all configuration choices every time, which can improve the efficiency of our algorithm.

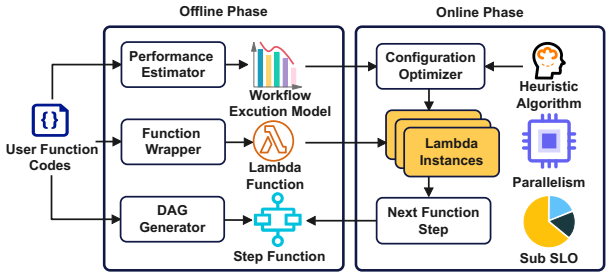

Fig. 8. The architecture overview of StepConf.

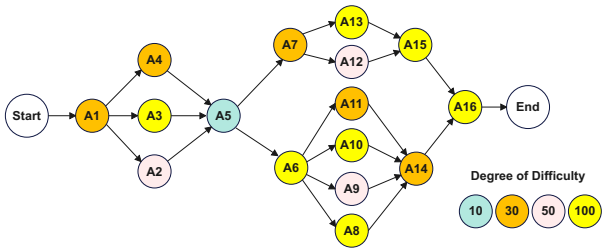

Fig. 9. The synthesized DAG workflow. Each node represents a function step. The different colors of the nodes represent tasks with different degree of difficulties and multi-core-friendly ratios.

V. Implementation

A. Framework Overview

We implement StepConf framework based on AWS Lambda and AWS Step Function. StepConf contains two main components: Performance Estimator and Configuration Optimizer.

As shown in Fig. 8, Performance Estimator captures the relationship among the configurations of each function, the corresponding cost, and performance. Configuration Optimizer makes configuration decisions dynamically for the workflow in real-time. At the beginning of each function step, Configuration Optimizer will use our algorithm to find the best configuration for the functions in the function step based on the SLO.

B. Function Step Configuration Optimizer

To run our algorithms to make per-step configuration in real time, we add a configuration optimizer function before each function step to run heuristics and make per-step configuration decision. Since AWS does not support determining the memory configuration when the function are invoked, we release function versions with different memory sizes in advance, and we can use aligns to specify the memory size for invocation. To implement inter-function parallelism, we use "map" operation with Step Function. It will automatically split the input payload data for different function instances during the operation. The degree of inter-function parallelism can be controlled by the shape of the input payload. At the same time, by adding "parallel" and "sequence" operations, we can construct most of the serverless function workflow logic.

C. Lambda Wrapper for Intra-function Parallelism

To achieve intra-function parallelism, we implement a wrapper class for invoking processes in a function. At first, the wrapper will get the configuration for intra-function parallelism from the function's payload. At the same time, it will get the input data from S3. When the data is ready, it will evenly split the data and start processes to call the user-defined function. We use `multiprocess.Manager()` to create `Queue()` for the communication between processes. At the end of the processes run, the wrapper will collect the return from each process, put them on S3, and return the obj keys to exit the function instance. To further accelerate performance, we re-write the `boto3` S3 interface using multi-threading, which squeezes out the network IO performance of Lambda functions. We will support more programming languages for Lambda Wrapper in future work, such as java, go, nodejs, etc.

D. The Running Process of StepConf

According to the description in Fig. 8, there are two main phases for the process of StepConf. In the offline phase, we ask the developers to provide the function code for each task with the workflow logic. Then, we will use Function Wrapper to deploy lambda functions, and DAG Generator will deploy the Step Function state machine. The Performance Estimator will fit the parameters of our workflow execution model through a few tests. During the online phase, when the workflow is running, StepConf will dynamically adjust the configuration with Configuration Optimizer to ensure the SLO.

VI. Experimental Evaluation

We design our evaluation to answer the following four research questions:

- Can Performance Estimator accurately predict the duration time and cost of the workflow in different configurations?
- Can Dynamic Configurator meet different workflow request SLOs through dynamic configuration?
- Compared with the static configuration strategy, does our dynamic strategy reduce the performance fluctuations?
- For each function step, how much overhead is brought by the configurator? Can the cost savings exceed the external overheads?

A. Experiment Setup

To evaluate StepConf, we built a video processing workflow similar to [23]. The workflow consists of 5 function steps. The workflow firstly uses OpenCV to pull the frames of the input video, then uses the DNN-based super-resolution algorithm to process the video frames, then respectively performs obj-detection and color-filtering, and finally encodes the processed video frames into video as output.

To evaluate general large DAG workflows, we synthesize a workflow based on a serverless benchmark suite [24]. The synthetic workflow is shown in Fig. 9. We control the task difficulty for each function step by configuring the times for the ALU test to repeat. In this way, we can simulate any workload by changing the multi-core-friendly ratio, task

Fig. 10. The fitting result of model for single-thread optimized function.

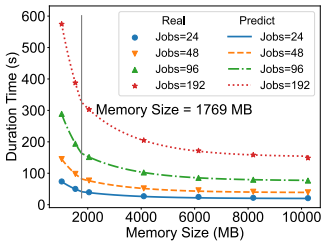

Fig. 11. The fitting result of model for multi-core-friendly function.

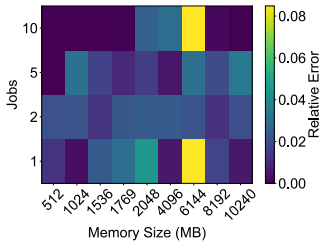

Fig. 12. Relative prediction error for single-thread optimized function.

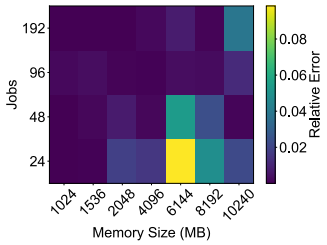

Fig. 13. Relative prediction error for multi-core-friendly function.

difficulty, and the amount of input and output data of the task workload. We created a workflow with similar methodology to [25] with 16 function steps for evaluation.

As shown in Algorithm 2, during the decision-making process of each function step, the latest completion timestamp the sub-graph of the previously calculated needs to be accessed. We use DynamoDB [20] as global cache to share this state among functions in the workflow. All our lambda functions choose `Python3.8` as the default runtime. For the video processing workflow, we use OpenCV to implement all the functions, and deploy them as docker images without configuring any provisioned concurrency.

We select the memory configurations in [512, 1024, 1536, 2048, 4096, 6144, 8192, 10240] MB, the degree of inter-function parallelism in [1, 2, 4, 8, 16, 32] and the degree of intra-function parallelism in [1, 2, 4, 8] for different function steps.

B. Evaluation for Performance Estimator

We use Performance Estimator to predict each workflow's duration time and cost in different SLOs for evaluation. Fig. 10 and 11 show the ground-truth and our prediction duration time of function steps under different task numbers and memory sizes. Fig. 10 represent the prediction accuracy of a single-thread optimized program and Fig. 11 represent the prediction accuracy of a multi-core-friendly program. We choose the optimal degree of intra-function parallelism for both of them. The highlight of our method is that we use different models and parameters to fit the parts with different multi-core friendliness and memory sizes. The average prediction errors are plotted in Fig. 12 and 13. The lighter the color, the greater the error of the predicted value relative to the ground truth. The maximum relative error of our models does not exceed 10%.

C. Evaluation for Configuration Optimizer

In this section, we evaluate the performance of Configuration Optimizer with our algorithm. The baselines we select are as follows.

- No-No: is generated from [26], which uses a static configuration strategy without considering inter or intra-function parallelism trade-off. A static strategy refers to making configuration decisions for all function steps before the workflow starts running. The configuration are unchanged during the workflow's run.
- Static: use the same static heuristic algorithm as No-No, but with same configuration factors (inter and intra-function parallelism) as StepConf (ours).
- Optimal: the offline optimal configuration with ignorance of fluctuations in performance.

We evaluate the cost and performance of StepConf by taking inter and intra parallelism into consideration, respectively.

- Intra-only: use the same dynamic strategy as StepConf (ours), considering intra-function parallelism but without considering inter-function parallelism trade-off.
- Inter-only: use the same dynamic strategy as StepConf (ours), considering inter-function parallelism but without considering intra-function parallelism trade-off.

Workflow performance under different SLOs: As comparison shown in the boxplots Fig. 14 and 15 and CDF plots in Fig. 16 and 17, our dynamic configuration strategy not only satisfies the request SLO better, but also brings smaller performance fluctuations. Due to the heavy dependency of the libraries, the gap between static and dynamic configuration in the fluctuation level is larger in video processing workflow than in synthetic workflow. The static method cannot guarantee the SLO for the requests. At the same time, our dynamic strategy can adjust the configuration in real-time to correct the error when the performance variation occurs during the running of workflow. These results reflect the advantages of our dynamic configuration strategy.

Cost saving and performance improvement: StepConf takes inter and intra-function parallelism into consideration. How do they influence the workflow's performance and cost? We compare the impact of intra and inter-function parallelism on workflow performance and cost, respectively. As baseline No-No does not utilize any function parallelism, we use it as a normalization unit. Compared with No-No, Fig. 18 shows that StepConf can improve performance by 5.32× faster under the same cost budget and save 40.9% cost with the same performance.

The main idea of StepConf is to dynamically configure each function step in the workflow through heuristic algorithms. However, real-time configuration for each function step will inevitably bring additional overhead. The question is that can the cost-saving covers the additional cost of configuration overhead? Fig. 19 to 21 show the experimental results of the overall average cost of the workflow under the conditions of

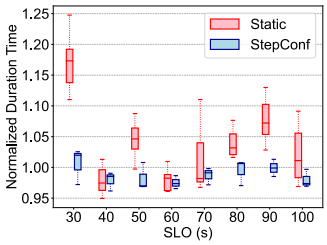

Fig. 14. Variation of synthetic workflow duration time.

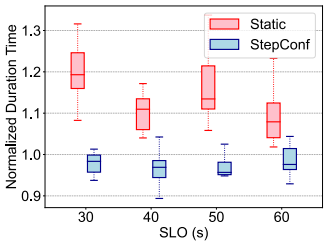

Fig. 15. Variation of video workflow duration time.

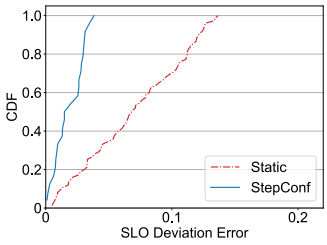

Fig. 16. Probability distribution of synthetic workflow duration time.

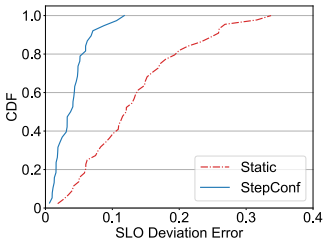

Fig. 17. Probability distribution of video workflow duration time.

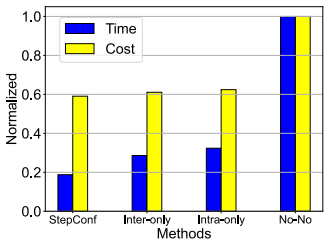

Fig. 18. Cost saving and performance improvement with none parallelism.

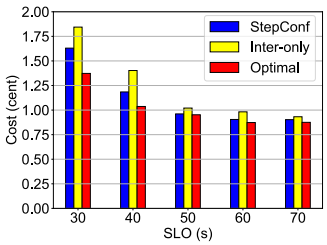

Fig. 19. Average cost of synthetic workflow under different SLOs.

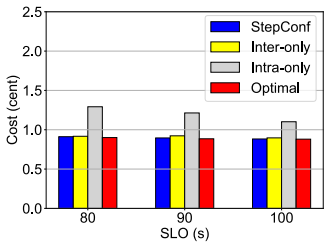

Fig. 20. Average cost of synthetic workflow under different SLOs.

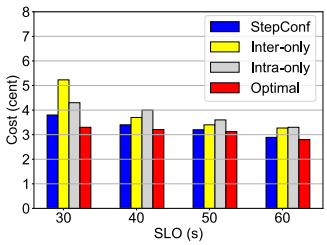

Fig. 21. Average cost of video workflow under different SLOs.

meeting different SLOs of the workflow. When the SLO is relatively small, our StepConf has a greater cost advantage than the inter-only or intra-only methods. When the SLO is relatively large, any inter or intra parallelism can achieve good results. Finally, compared with the offline optimal method, the cost of StepConf is at most 16.48% higher on our testbed. Although the heuristic algorithm of StepConf cannot ensure the globally optimal configuration, it can also guarantee the effect of optimization at most times.

VII. Related Work

Resource scheduling in cloud cluster: Provisioning cloud resources and scheduling jobs to meet different user requirements are hot research topics in recent years [27]. Zhang et al. design an online and QoS-aware data-driven cluster manager for interactive cloud microservices [28]. Mao et al. implement deep reinforcement learning for efficiently scheduling data processing jobs on distributed compute clusters [29]. Li et al. dynamically configure the scale of VMs to achieve a cost-efficient solution for MapReduce workloads [30]. Romero et al. develop an automated model-less system for distributed inference serving, where developers specify the performance and accuracy requirements for their applications without needing to select a specific model-variant for each query [31].

Performance and cost optimization for serverless: Akhtar et al. optimize the configuration of function chains by using a statistical learning method with fewer tests [32]. Elgamal et al. jointly optimize function placement and function size to save cost [33]. Eismann et al. use mixture density models to predict the execution time distribution of a function to optimize the costs of serverless workflows [14]. Carver et al. design a FaaS computing framework for DAG workflows with data locality optimized [12]. Bhasi et al. propose a workflow-aware serverless function resource adaptive autoscaler [34]. Zhang et al. introduce a new scheduling problem to determine the ideal task launch times for serverless data analytics [35]. Kijak et al. present formulate a deadline-aware function resource scheduling problem for scientific workflows and then propose a configuration strategy for cost saving [26]. Lin et al. describe the function workflows as none-DAG pattern, which contains the cycles and self-loop. They propose a probability refined critical path algorithm to solve both the problem of best cost under performance constraint and best performance under budget constraint [25]. Although we share similar idea, the main advantage of our work is that we adopt a dynamic configuration strategy and aim to guarantee the SLO of every workflow's request. Due to the degradation of function performance, existing works fail to guarantee the request SLOs. In addition, we take the inter and intra-function parallelism into consideration, which can greatly improve the performance of the workflows.

VIII. Conclusion

We present StepConf, a dynamic resource configuration framework for serverless function workflows. StepConf takes advantage of piece-wise fitting models, making precise cost and performance predictions. We consider inter and intra-function parallelism and design a per-step dynamic configuration strategy that guarantees end-to-end workflow SLOs. We propose an online heuristic algorithm to find a suitable solution in polynomial time. Compared with existing static strategies, StepConf can improve performance by 5.32× faster under the same cost budget and save 40.9% cost with the same performance. Compared with the offline optimal method, the cost of StepConf is at most 16.48% higher on our testbed.

REFERENCES

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

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

[3] "Openfaas - serverless functions made simple." https://www.openfaas.com.

[4] "Apache openwhisk - open source serverless cloud platform." https://openwhisk.apache.org.

[5] "Amazon aws lambda." https://aws.amazon.com/lambda/.

[6] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proc. of USENIX NSDI*, pp. 363–376, 2017.

[7] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proc. of ACM SoCC*, pp. 263–274, 2018.

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

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

[10] V. Sreekanti, H. Subbaraj, C. Wu, J. E. Gonzalez, and J. M. Hellerstein, "Optimizing prediction serving on low-latency serverless dataflow," *arXiv preprint arXiv:2007.05832*, 2020.

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

[12] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proc. of ACM SoCC*, pp. 1–15, 2020.

[13] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proc. of ACM SoCC*, pp. 13–24, 2019.

[14] S. Eismann, J. Grohmann, E. Van Eyk, N. Herbst, and S. Kounev, "Predicting the costs of serverless workflows," in *Proc. of ACM/SPEC ICPE*, pp. 265–276, 2020.

[15] A. Casalboni, "Aws lambda power tuning." https://github.com/alexcasalboni/aws-lambda-power-tuning, 2020.

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

[17] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. of USENIX ATC*, pp. 205–218, 2020.

[18] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. of ACM ASPLOS*, pp. 467–481, 2020.

[19] "Amazon s3: Object storage built to retrieve any amount of data from anywhere." https://aws.amazon.com/s3.

[20] "Databases on aws." https://aws.amazon.com/products/databases.

[21] "Aws lambda power tuning." https://aws.amazon.com/lambda/pricing, 2021.

[22] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506–521, 1996.

[23] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," in *Proc. of ACM SoCC*, pp. 1–17, 2021.

[24] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proc. of ACM SoCC*, pp. 30–44, 2020.

[25] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 615–632, 2020.

[26] J. Kijak, P. Martyna, M. Pawlik, B. Balis, and M. Malawski, "Challenges for scheduling scientific workflows on cloud functions," in *Proc. of IEEE CLOUD*, pp. 460–467, 2018.

[27] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *Proc. of USENIX NSDI*, pp. 469–482, 2017.

[28] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *Proc. of ACM ASPLOS*, pp. 167–181, 2021.

[29] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. of ACM SIGCOMM*, pp. 270–288, 2019.

[30] Y. Li, F. Liu, Q. Chen, Y. Sheng, M. Zhao, and J. Wang, "Marvelscaler: A multi-view learning based auto-scaling system for mapreduce," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2019.

[31] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "{INFaaS}: Automated model-less inference serving," in *Proc. of USENIX ATC*, pp. 397–411, 2021.

[32] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "Cose: Configuring serverless functions using statistical learning," in *Proc. of IEEE INFOCOM*, pp. 129–138, 2020.

[33] T. Elgamal, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *Proc. of IEEE Symposium on Edge Computing*, pp. 300–312, 2018.

[34] 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. of ACM SoCC*, pp. 153–167, 2021.

[35] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: Nimble task scheduling for serverless analytics.," in *Proc. of USENIX NSDI*, pp. 653–669, 2021.