Joint Optimization of Parallelism and Resource Configuration for Serverless Function Steps

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

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 multi-step workflow. The developers are required to set proper configurations for functions to meet service level objectives (SLOs) and save costs. However, developing the configuration strategy is challenging. This is mainly because the execution of serverless functions often suffers from cold starts and performance fluctuation, which requires a dynamic configuration strategy to guarantee the SLOs. In this paper, we present StepConf, a framework that automates the configuration as the workflow runs. StepConf optimizes memory size for each function step in the workflow and takes inter and intra-function parallelism into consideration, which has been overlooked by existing work. StepConf intelligently predicts the potential configurations for subsequent function steps, and proactively prewarms function instances in a configuration-aware manner to reduce the cold start overheads. We evaluate StepConf on AWS and Knative. Compared to existing work, StepConf improves performance by up to 5.6× under the same cost budget and achieves up to a 40% cost reduction while maintaining the same level of performance.

Index Terms—Serverless Computing, Resource Management, Resource Configuration, Function Workflow.

1 Introduction

Function-as-a-Service (FaaS) is a new paradigm for serverless computing that allows developers to run code in the cloud without having to maintain and operate cloud resources [1], [2]. Developers need only submit function code to FaaS platforms, where compute resources are provisioned and functions are executed seamlessly. Therefore, developers only need to focus on business logic, which accelerates application development progress and saves operational costs. With FaaS platforms like AWS Lambda [3], developers can rapidly leverage hundreds of CPU cores by invoking massive functions simultaneously.

Benefiting from the fine-grained, high elasticity of FaaS, many applications have been developed based on serverless functions, including video processing [4], [5], [6], machine learning [7], [8], [9], [10], code compilation [11], big-data analytic [12], [13], [14], [15], [16], etc. To migrate applications to FaaS platforms, developers need to decouple monolithic applications into multiple functions, resulting in complex multi-step serverless workflows. However, developers face challenges in setting proper configurations for workflows to optimize the cost and ensure performance. We summarize these challenges as follows:

Vague Impact of Resource Configurations: The cost and performance of functions in FaaS depend heavily on user-configured resource parameters. However, due to the unique resource allocation and pricing mechanisms of FaaS platforms, understanding the impact of resource configuration can be challenging. Therefore, it is difficult to determine the appropriate resource parameters to achieve high performance and low cost. As a result, developers often struggle to determine the optimal resource configuration for their workflows [17], [18], [19].

Performance Fluctuations in Workflows: FaaS frequently depend on external storage services to enable extensive data exchange in workflows [20], [21]. Nevertheless, these external services demonstrate considerable variations in data transmission delay, ranging from a few hundred milliseconds to a couple of seconds. Additionally, function cold starts prevent concurrent mapping invocations from starting simultaneously, leading to different mapping delays. These factors introduce fluctuations in the performance of function steps, rendering it difficult to ensure end-to-end SLOs for workflows concerning configuration optimization.

Exponential Growth of Configuration Space: FaaS app developers are required to configure several parameters for each function. As the number of functions in the workflow grows, the decision space for parameters grows exponentially, making it difficult to find the optimal configuration. Adjusting the configuration according to different SLOs further complicates the resource configuration problem.

Challenge in Optimizing Parallelism of Function Step: The stateless nature of FaaS functions combined with their robust scalability simplifies and enhances parallel execution. FaaS workflows are particularly suitable for both inter-

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

Manuscript received xxxx xx, 2023; revised xxxx xx, 2023.

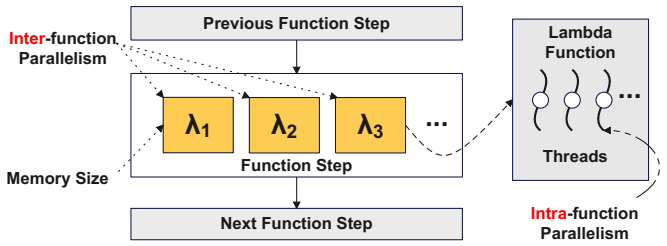


Fig. 1: Resource configuration parameters in a function step. The yellow blocks represent concurrently running function instances, where each function instance consists of multiple processes to achieve intra-function parallelism.

function and intra-function parallelism. A function step can not only implement inter-function parallelism via concurrent instances, but it can also execute intra-function parallelism using multiple processes within a single function instance. Therefore, besides the function memory size, it is equally critical to optimize the parallelism within each function step in the workflow, which is overlooked by existing works [22], [23], [24].

To better solve these challenges, in this paper, we present the design, implementation, and evaluation of StepConf, an SLO-aware dynamic cost optimization framework for multi-step serverless workflows. Our approach is an online solution that dynamically determines the configuration for each function step before its execution, jointly optimizing inter-function parallelism, intra-function parallelism, and function memory size, as shown in Figure 1.

In this way, we can not only improve resource efficiency through optimized parallelism, but also optimize the configuration step-by-step in real time to correct workflow progress, mitigate the impact of function performance variations, and ensure workflow SLOs while reducing costs. We also developed a function prewarming mechanism to mitigate function cold start to reduce performance fluctuations.

First, we perform extensive measurements on AWS Step Functions to analyze the underlying service mechanism of serverless workflows. Our discussion indicates that performance fluctuations are primarily driven by the data transmission of external storage services and the cold start of functions. To tackle this challenge, we have developed a model in performance estimator that leverages historical data, applying piece-wise fitting for both single-core and multi-core scenarios. Meanwhile, we use statistical quantile regression models to estimate the mapping delays and data transmission delays of functions. These approaches resolve the vague impact of resource configuration on performance and provide accurate estimates of performance and cost for online dynamic configuration optimization.

Second, we formally model the workflow resource configuration optimization problem and show that it is NP-hard. To this end, we transform the problem into a decision problem for each function step and utilize a heuristic optimization algorithm based on the critical path to satisfy real-time requirements at low costs. It is an online method that dynamically optimizes the configuration of the entire workflow step by step. We employ the Configuration Optimizer to execute this algorithm. It achieves optimal resource utilization by dynamically optimizing each step of the workflow in real-time while meeting SLO requirements.

Third, we integrate an advanced workflow engine into StepConf. This engine efficiently coordinates various functions according to user-defined workflows and closely integrates with the Configuration Optimizer, enabling dynamic configuration optimization throughout the execution of the workflow. Additionally, we incorporate a Configuration-Aware function prewarming mechanism into the workflow engine. It utilizes the dependency within the workflow to proactively prewarm function instances. By accurately predicting the configuration needs for the next step and prewarming the requisite number and size of function instances in advance, this mechanism significantly reduces the likelihood of cold starts in the following function steps. Moreover, we design a highly scalable Function Manager, which not only facilitates efficient parallel processing of function steps but also enhances the flexibility and scalability of StepConf, enabling it to support a wide range of FaaS platforms, including open-source and commercial platforms.

Finally, extensive experimental results show that StepConf improves performance by up to 5.6× under the same cost budget and achieves up to a 40% cost reduction while maintaining the same level of performance compared with baselines.

Considering that this work is an extension based on our previous work [25], we highlight the key improvements as follows.

- We enhance the accuracy of our performance estimation model by integrating a quantile regression model, which effectively predicts mapping and data transmission delays, thus significantly improving the overall precision of our performance estimation.
- We design a new Configuration-Aware function prewarming mechanism to StepConf, which leverages the workflow dependencies and performance models to initialize the required function instances in advance for the next function step, reducing the mapping delay caused by cold starts.
- We reconstruct the architecture of StepConf to support open-source FaaS platforms. We design and implement a workflow engine and adapt it to different platforms by adding the Function Manager.
- We expand the experimental evaluation by introducing a machine learning workflow and comparing it with additional baselines and evaluation metrics to better demonstrate the advantages of StepConf.

The rest of this paper is organized as follows: The background and motivation are introduced in Section 2; Section 3 describes how StepConf establishes its performance estimation model; Section 4 details the definition of the configuration optimization problem and the design of heuristic algorithms for its resolution; Section 5 presents the function prewarming mechanism; Section 6 discusses the design and implementation details of StepConf's system components; Section 7 shows the experimental evaluation of StepConf. The related work is reviewed in Section 8, with the conclusion of this study presented in Section 9.

2 Background And Motivation

In this section, we first introduce the background of function workflows. Then, we summarize the shortcomings of existing work and discover two key factors they overlook. Finally, we discuss the design motivation behind workflow configuration optimization.

2.1 Configuration for Multi-step Serverless Workflows

Many real-world serverless applications are implemented as multi-step function workflows, where incoming user requests invoke a set of serverless function steps, and each function step performs a specific task while coordinating with other function steps to complete the entire workflow. These function steps can be executed either serially or in parallel to improve processing performance.

However, optimizing the resource configuration of functions within serverless workflows presents a major challenge for developers, as previously discussed. There has been some previous work conducted in this area. For instance, AWS Lambda Power Tuning Tools [18] provides a tool to help developers optimize serverless function configurations, reducing costs while achieving performance goals. However, this tool is only suitable for optimizing the configuration of a single node and does not apply to serverless workflows. COSE [22] models and optimizes resource configuration for chain-based workflows. It intelligently collects samples using Bayesian statistical methods and predicts the cost and execution time of serverless functions under different configurations. It also uses integer linear programming to solve the optimization problem, meeting the SLO of the user's workflow. However, this work does not consider complex DAG workflows. ORION [23] establishes a performance model for DAG-style function workflows and proposes a method for co-locating multiple parallel function calls within a single Virtual Machine (VM). It optimizes function resource configuration to meet the E2E latency requirements of the workflow.

However, these works have shortcomings. They all overlook the importance of jointly optimizing parallelism and resource allocation in the function steps (Section 2.2), and the need to dynamically configure optimized workflows may lead to difficulties in guaranteeing SLO (Section 2.3). This gives us the motivation for our research.

2.2 Jointly Optimize Parallelism in Function Steps

In the FaaS environment, choosing the appropriate memory size for functions is crucial. However, beyond the configuration dimension of resource specifications, the parallelism of function steps is also a very important dimension.

Firstly, we can utilize the parallelism between concurrent function instances (inter-function parallelism). For example, in a video processing workflow, we can simultaneously invoke different functions to perform various tasks for processing video images, as shown in Figure 14. Although each function has limited computational capability, we can achieve exceptional performance through high parallelism among function instances.

However, higher parallelism between functions does not always mean benefits. We evaluated the performance of

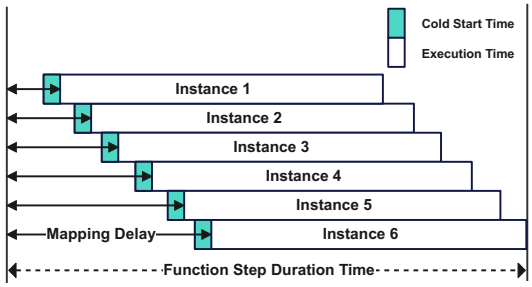


Fig. 2: The working process of a function step. In a function step, concurrent instances start running one by one, with each instance having a different mapping delay.

Step Function workflows on AWS by decomposing the total computational delay of a function step into two parts: mapping delay and execution time. Figure 2 shows the process of AWS Step Function executing a function step where different concurrent instances in a function step are not started simultaneously. Thus, they have different mapping delays. The time from the start of the function step to the start of the last invoked function instance is referred to as the mapping delay. Mapping delay is caused by the overhead of function cold starts and is related to the elastic capabilities of the FaaS platform.

We created function steps with different function execution time and invoked them with different inter-function parallelism on the AWS Step Function. As shown in Figure 3, the mapping delay of the function steps is relatively small when the parallelism between functions is less than 30. However, the mapping delay increases rapidly when it exceeds 40. As shown in Figure 4, continuously increasing the inter-function parallelism does not further speed up the computation. The reason is that the mapping delay becomes a bottleneck. Therefore, we need to optimize the selection of function memory size and inter-function parallelism between functions to achieve cost efficiency.

Besides parallelism between function instances, there is also intra-function parallelism, as illustrated in Figure 1. Tasks within a function instance can be parallelized in a multi-process manner to achieve parallelism. Therefore, applying intra-function parallelism strategies can undoubtedly improve the resource utilization and computational efficiency of functions, especially for programs not optimized for multi-core processing. To demonstrate this, we prepared a program that can only utilize single-core capabilities and observed its performance under different levels of intra-function parallelism. As can be seen in Figure 5, AWS allocates different numbers of available vCPUs for functions with various memory sizes. Increasing intra-function parallelism in situations with multiple CPU resources available allows functions to utilize more CPU resources to enhance performance. According to the AWS Lambda official documentation [3], functions have only single-core CPU resources when memory is less than 1769MB, and multi-core CPU resources are available only when exceeding 1769MB.

To delve deeper into how increasing memory size affects the performance of functions with varying degrees of multi-core friendliness, we deploy a benchmark function on AWS Lambda that assesses whether each large number in a list

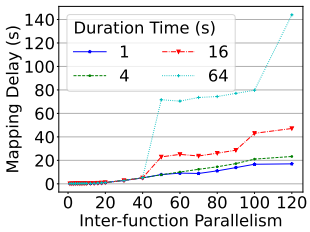


Fig. 3: Mapping delay of a function step with different function duration time and inter-function parallelism.

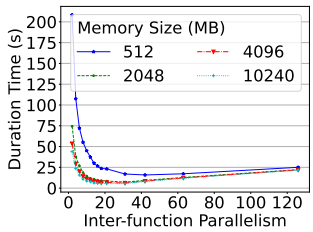


Fig. 4: Duration time of a function step with different memory size and inter-function parallelism.

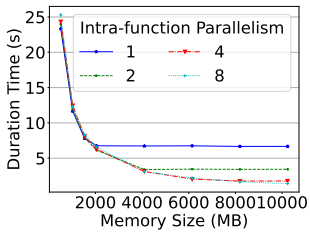


Fig. 5: Duration time of a function step with different memory size and intra-function parallelism.

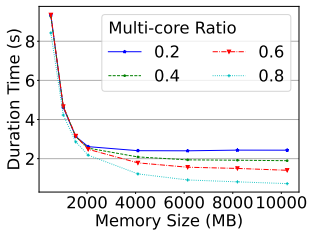


Fig. 6: Duration time of a function with different memory size and multi-core ratio.

is a prime number. We develop both single-threaded and multi-core optimized versions of this program. The multi-core ratio is set as a task ratio to fully exploit multi-core resources. The higher this ratio, the more the program is optimized for multi-core processing. We measure the function's performance across various multi-core ratios and memory sizes. The results, as shown in Figure 6, indicate that the performance of single-thread optimized programs ceases to improve beyond a certain memory threshold. In contrast, the duration of the multi-core optimized programs proportionally decreases as memory increases. This observation leads us to the conclusion that it is essential to distinguish between single-core and multi-core scenarios.

As a result, in addition to inter-function parallelism, choosing the appropriate level of intra-function parallelism is also key to improving workflow performance and reducing costs. Jointly optimizing parallelism while optimizing the resource configuration of function steps is also very important.

2.3 Performance Fluctuations in Function Steps

Performance fluctuations in function steps during execution pose a challenge to ensuring end-to-end SLOs for workflows. Through our analysis, we find that the primary causes of performance fluctuation are the variability in data exchange overhead and the unpredictable mapping delay caused by function cold starts.

As shown in Figure 2, due to the possibility of cold starts in different function instances within the function step, the mapping delay exhibits its own variability. Additionally, in the FaaS architecture, the separation of storage and computation necessitates the use of remote storage services to facilitate data exchange between functions [20], [21]. Unfortunately, this dependency often leads to significant latency, varying from several hundred milliseconds to a few seconds. We also notice significant variations in the performance of these third-party storage services. To address this issue, we conduct a series of tests to evaluate the performance of AWS S3 storage within the AWS Lambda environment.

We use Python lambda functions and the AWS Boto3 SDK to download objects of different sizes from S3 buckets. We measure the total data transmission delay while varying the number of objects and their size. The results show different distribution patterns, as illustrated in Figure 7. We observe that the IO performance of S3 exhibits characteristics of a long-tail distribution. The variation between the fastest and slowest cases can often be several times. This phenomenon increases the complexity of optimizing workflow performance.

Therefore, it is necessary to dynamically adjust the configuration of function steps within the workflow in real-time, promptly correct deviations in workflow progress.

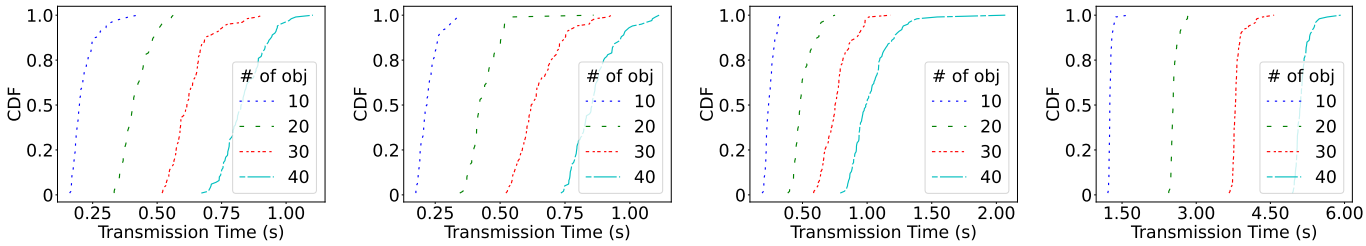
2.4 Design Principles

Based on the above analyses and insights, we formulate the following core design principles for the system:

- **Joint Optimization of Memory and Parallelism Configurations for Function Steps:** To achieve the best cost-performance ratio, it is crucial to simultaneously consider and optimize the memory configuration and parallelism settings of function steps.
- **Developing a Performance Estimation Model:** Developing a model capable of accurately predicting performance under different configurations is essential. This will provide solid data support for configuration choices in a serverless environment, helping to resolve the ambiguity in configuration awareness.
- **Dynamic Online Configuration:** Given the variability in performance across different steps of a workflow, online dynamic configuration optimization is vital. This strategy involves dynamically adjusting configuration choices before the execution of each function step to ensure the achievement of workflow performance goals.
- **Minimizing Function Cold Starts:** Implementing strategies to minimize or avoid cold starts of function instances is necessary to enhance the execution efficiency of workflows. This not only reduces latency but also improves resource utilization efficiency.

3 Estimating Performance and Cost under Different Configurations

In this section, we introduce how to establish a performance estimation model to estimate the duration time and cost of a function step under different configurations (parallelism and resource configuration).



(a) Obj size of 0.1 MB. (b) Obj size of 0.5 MB. (c) Obj size of 1 MB. (d) Obj size of 10 MB.

Fig. 7: CDF plots of data transmission time with different object sizes.

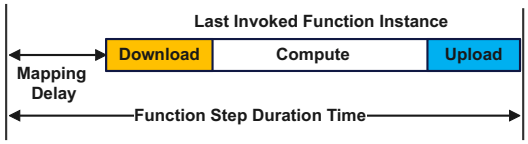


Fig. 8: Decomposition of function step duration time. When the last invoked function instance completes its execution, the process of the function step ends.

3.1 The Importance of Performance Estimation

To optimize the configuration of the function step, we need to obtain the performance and cost of the function step under different configurations. One direct method is to conduct comprehensive profiling tests. However, performing profiling tests on all possible configuration combinations in advance is both time-consuming and costly. We hope to estimate the relationship between different configuration parameters, performance, and cost using a smaller sample of profiling tests.

3.2 Decomposition of Function Step Duration Time

To accurately estimate the performance of function steps, we decompose the duration time of function steps. The execution process of a function step corresponds to the completion of all tasks in that function step during the execution of a workflow. As we consider inter-function and intra-function parallelism within each function step, the entire process of a function step is equivalent to the execution of multiple concurrent instances of a function, where each instance utilizes multiprocessing for parallel computation.

Through our tests, we find that concurrently invoked function instances are not started at the same time, and these concurrent instances experience different mapping delays. The mapping delay of a function instance is the waiting time from the start of the function step until the function instance begins to execute. The mapping delays of different concurrent function instances within the same function step form a stair-like distribution, as shown in Figure 2. This phenomenon is caused by the cold start time required for the elastic scaling of function instances, where function instances are created one by one rather than all at once. In addition, the creation of function sandboxes and initialization of function code, which also contribute to the mapping delay.

Since the tasks of concurrent instances within a function step are evenly distributed, the execution times are similar. To this end, we can define the mapping delay of a function step as the mapping delay of the latest invoked function instance. Therefore, the duration time of a function step can be divided into the mapping delay plus the function execution time.

As shown in Figure 8, the total time for a function step consists of the mapping delay of the last invoked function and the execution time of its function instance. The execution time of each function instance in the function step mainly includes three parts: data download, task computation, and data upload. Let's denote a certain function step as v_i, its end-to-end total delay as t_i, the mapping delay as t_i^m, the total delay for data download and upload as t_i^d and t_i^u respectively, and the computation delay of the function instance as t_i^c, and we have: $t_i = t_i^m + t_i^d + t_i^c + t_i^u$.

Among these three parts, the time for downloading and uploading data is usually determined by the function's network bandwidth and is independent of the function's computational complexity. The computation time depends on the function's computational complexity and the resources used.

Based on previous analysis, data transmission delay and mapping delay have significant fluctuations. On the other hand, the fluctuation of computation latency is relatively small. Therefore, we need to employ different estimation methods for these components.

3.3 Piece-wise Fitting Model for Computation Latency

To accurately predict the computation latency of functions, we propose a piece-wise fitting model. Since the duration of function computation is usually influenced by the number of tasks in function instances and the memory size, we consider these two key factors in the model. Let γ_i represent the total number of tasks in a function step. Then, based on the inter-function parallelism p_i, we define $\overline{\gamma} i = \frac{\gamma i}{p_i}$, which represents the number of tasks allocated to each function instance.

Based on the experimental data shown in Figure 6, we find that the execution time of functions is inversely proportional to the memory size. Since performance is the reciprocal of time, this indicates that the multi-core performance of function instances is nearly linearly related to the memory size. At the same time, the performance of functions is also influenced by the internal parallel optimization of function code. We use m_s to represent the memory size of

single-core resources allocated by the cloud service provider (where s denotes single-core), and δ_s represents the relative performance. We define $\min(\frac{m_i}{m_s}, q_i)$, where q_i represents the index of internal parallelism in the function. Here, $\frac{m_i}{m_s}$ represents the multiple of resources allocated to the function compared to the maximum multiple achievable by a single core, and q_i represents the multiple of performance improvement achievable by using internal function parallelism compared to a single thread. Taking the smaller value of these two factors represents the upper limit of the multi-core acceleration ratio for a single-thread optimized function with internal parallelism.

It is worth noting that if the program has already implemented multi-core optimization, the impact of internal function parallelism will be weakened. In this case, the inverse relationship between execution time and memory size will change. Therefore, we use an exponential function to improve prediction accuracy.

Thus, we design a piece-wise fitting approach. For programs applicable to multi-core scenarios (i.e., case 1), we use an exponential function for fitting; while for programs that can only utilize single-thread performance or have memory sizes lower than m_s (i.e., case 2), we use an inverse function for fitting:

$$t_i^c = \begin{cases} (a_{i,1} \cdot \overline{\gamma} i + b_{i,1}) e^{-a_{i,2} \cdot \min(\frac{m_i}{m_s}, q_i)} + \phi_{i,1}, & \text{case 1} \\ \frac{a_{i,2} \cdot \overline{\gamma} i}{\delta s \cdot \min(\frac{m_i}{m_s}, q_i) + b_{i,2}} + \phi_{i,2}, & \text{case 2} \end{cases} \quad (1)$$

In this formula, $a_{i,1}, a_{i,2}, b_{i,1}, b_{i,2}, \phi_{i,1}, \phi_{i,2}$ are model parameters obtained through data training.

3.4 Quantile Regression Model for Varying Delays

Considering the inherent variability in data transmission and mapping delays, making precise predictions poses a significant challenge. To address this issue, we adopt a quantile regression model for prediction. This model is particularly suitable for predicting response variables like data transmission delay and mapping delay at specific quantiles. Imagine we have a dataset containing numerous variables observed under various parameters. Here, Y represents the response variable, while $\mathbf{X} = [X_1, X_2, \ldots, X_n]$ represents a set of explanatory variables.

To more accurately capture the potential nonlinear relationships between $\mathbf{X}$ and Y, our quantile regression model considers not only the effect of each individual X_i on Y, but also includes the square terms of these variables and their possible interactions. The model can be expressed as:

$$\begin{aligned} Q_{Y|\mathbf{X}}(\omega) = \beta_0(\omega) + \sum_{i=1}^{n} \beta_i(\omega) X_i + \\ \sum_{i=1}^{n} \gamma_i(\omega) X_i^2 + \sum_{i<j} \delta_{ij}(\omega) X_i X_j \end{aligned} \quad (2)$$

Where ω is quantile, $\beta_0(\omega)$ is the intercept term, and $\beta_i(\omega)$, $\gamma_i(\omega)$, and $\delta_{ij}(\omega)$ respectively represent the coefficients for the linear terms, square terms, and interaction terms. This model structure allows us to capture and predict the distribution of the response variable Y at the quantile level, while considering the nonlinear relationships and interaction effects among the explanatory variables.

These coefficients are estimated by minimizing the quantile loss function, which focuses on absolute deviations and assigns different weights to errors above and below the ω quantile. The estimated parameters are used to predict data transmission time and mapping delay at the specific quantile ω. We use the number of objects and its size as parameters for predicting data transmission delay, and the number of functions with a cold start for predicting mapping delay. Although the quantile regression model does not directly estimate the complete probability distribution, it does provide significant insights into the conditional distribution of delays at the chosen quantile level.

3.5 Cost Model 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 [17]. 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. For a function step, given a intra-function parallelism p_i, then we have the cost of function step $c_i = p_i \cdot (t_i \cdot m_i \cdot \mu_0 + \mu_1)$.

4 Dynamic Optimization of Function Step Configuration

In this section, we introduce how to dynamically determine the configuration for each function step.

4.1 Problem Formulation

A function workflow consists of a series of different function steps executed in sequence. We represent a serverless workflow using a directed acyclic graph (DAG) $G = (V, E)$. The vertices $V = v_1, v_2, ..., v_n$ represent the n function steps within the workflow. Vertices with an in-degree of 0 represent the starting point of the workflow, which corresponds to the first function step to be executed. Vertices with an out-degree of 0 represent the endpoint of the workflow, which corresponds to the last function step to be executed. The edges $E = \widehat{v_i v_j} \mid 1 \leq i \neq j \leq N$ represent dependencies between function steps, where $\widehat{v_i v_j}$ denotes the edge $v_i \rightarrow v_j$, meaning that the operation of function step v_j depends on v_i.

For function step v_i, it will only begin execution once all its dependencies are completed. Let $L \in \mathcal{L}$ represent the set of vertices from the start to the endpoint. We define the execution time of function step v_i as t_i and the completion time of the function workflow as T. It is easy to see that the completion time T depends on the longest running path and can be expressed as $T = \max_{L \in \mathcal{L}} \sum_{v_i \in L} t_i$.

Furthermore, we define the cost of function step v_i as c_i, and the total cost of processing the workflow can be calculated using $C = \sum_{v_i \in V} c_i$.

The memory size of all parallel functions in each function step is the same as the memory configuration of the function step. Let $\mathcal{M}$ represent the set of memory configurations provided by the cloud platform, and $\mathcal{P}$ represent the set of optional inter-function parallelism. We use $\mathcal{Q}$ to

represent the set of available optimal intra-function parallelism. We represent the configuration of function step v_i as $\theta_i = (m_i, p_i, q_i)$. We can define the configuration settings of function step v_i as:

$$\Theta_i = (m_i, p_i, q_i) \mid m_i \in \mathcal{M}, p_i \in \mathcal{P}, q_i \in \mathcal{Q}. \tag{3}$$

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

$$x_i(\theta) = \begin{cases} 0, & \text{function step } v_i \text{ is not configured as } \theta \\ 1, & \text{function step } v_i \text{ is configured as } \theta \end{cases}. \tag{4}$$

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

$$\sum_{\theta \in \Theta_i} x_i(\theta) = 1, \forall v_i \in V. \tag{5}$$

Our goal is to optimize the end-to-end performance of the workflow and minimize cost while satisfying the workflow's Service Level Objective (SLO). Let $\mathcal{S}$ represent the SLO of the workflow request. We have the following SLO constraint:

$$T = \max_{L \in \mathcal{L}} \sum_{v_i \in L} \sum_{\theta \in \Theta_i} t_i(\theta) \cdot x_i(\theta) < \mathcal{S}. \tag{6}$$

Our Function Workflow Configuration Problem (FWCP) can be formulated as:

$$\begin{aligned} \min C = & \min \sum_{v_i \in V} \sum_{\theta \in \Theta_i} c_i(\theta) \cdot x_i(\theta) \\ & \text{subject to: } (3)(4)(5)(6). \end{aligned} \tag{7}$$

In summary, the FWCP aims to find the optimal configuration for each function step in the workflow to minimize the total cost while ensuring that the end-to-end performance meets the given SLO constraint. The problem is formulated using a series of equations and constraints that take into account the execution time, cost, and configuration of each function step, as well as the dependencies between function steps in the workflow. Main notations are listed in Table 1.

4.2 Computation Complexity

Theorem 1 shows that the problem (7) is NP-hard, which makes it impractical to find the optimal configuration.

Theorem 1. *Problem **FWCP** (7) is NP-hard.*

Proof. We construct a polynomial-time reduction to (7) from the multiple-choices knapsack problem (8)(9), a classic optimization problem which is known to be NP-hard.

$$\max \sum_{i=1}^{m} \sum_{j \in N_i} \mu_{ij} x_{ij} \tag{8}$$

$$\text{subject to : } \begin{cases} \sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} \leq C \\ x_{ij} \in \{0,1\}, 1 \leq i \leq m, j \in N_i \\ \sum_{j \in N_i} x_{ij} = 1, 1 \leq i \leq m \end{cases}. \tag{9}$$

Given an instance $A = (m, \mu_{ij}, w_{ij}, C, x_{ij})$ of the knapsack problem, we map it to an instance of the (7) 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, the multiple-choices knapsack problem can be treated as a special case of (7). Given the NP-hardness of the multiple-choices knapsack problem, (7) must be NP-hard as well. □

TABLE 1: Main Notations

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

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

As shown above, we need to reduce the computation complexity and propose a practical algorithm to configure each function step efficiently.

Note that (6) 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

$$\forall L \in \mathcal{L}, \sum_{v_i \in L} s_i \leq \mathcal{S}. \tag{10}$$

By doing this, we relax the constraint (6) by:

$$t_i = \sum_{\theta \in \Theta_i} t_i(\theta) \cdot x_i(\theta) < s_i. \tag{11}$$

Then we can reduce the problem (7) of optimizing a DAG to optimize the configuration of each function step v_i with sub-SLO constraints, which is formulated as follows:

$$\begin{aligned} \min c_i = & \min \sum_{\theta \in \Theta_i} c_i(\theta) \cdot x_i(\theta) \\ & \text{subject to: } (4)(11). \end{aligned} \tag{12}$$

In the following discussion, we will demonstrate how to allocate sub-SLOs to each function step. Based on the critical path method [26], our approach identifies the longest path in the workflow as the critical path. The execution time of each node on this critical path is considered its corresponding weight. These weights then serve as the basis for fairly distributing the SLO among the nodes (function

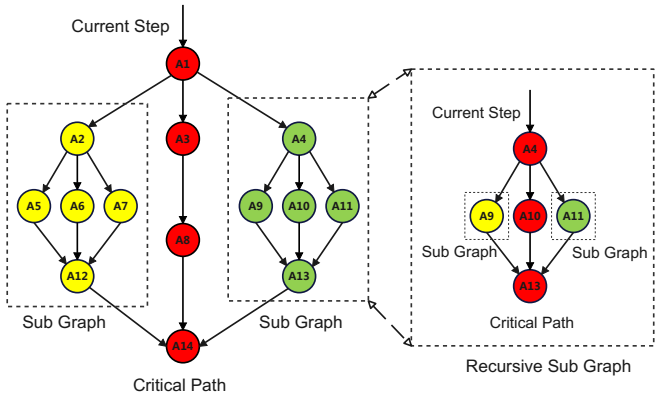


Fig. 9: Algorithm description diagram.

steps). This allocation mechanism is visually represented in Figure 9, where the critical path is assumed to be the red path in the diagram (A1-A3-A8-A14). We can easily allocate sub-SLOs to all nodes on the critical path and find the optimal configuration for the current node (A1). Subsequently, we identify two sub-graphs connected to the critical path (yellow and green). As the workflow progresses and encounters the starting nodes of the sub-graphs (A4), we can recursively apply our method. With the workflow's execution, the size of the sub-graphs decreases (A9 and A11), finally achieving online dynamic resource allocation optimization for each node in the workflow.

However, in our online problem, we are unable to know the configuration choices and their runtime for future functional steps, making it challenging to determine the weights to find the critical path. Based on our insights, each functional step has a most cost-effective configuration, defined by the ratio of performance to cost. We rank configurations according to their cost-effectiveness and observe that configurations close to the most cost-effective one also demonstrate relatively high cost-effectiveness. Building on this, we utilize performance estimation model to identify the most cost-effective configuration for each node, using its duration to set weights. This approach allows us to approximate the sub-SLO for each function step closer to the most cost-effective configuration choice, even when we cannot pre-determine the best weight distribution. This strategy helps in enhancing the overall computational efficiency of the workflow and reducing costs.

In our problem, the performance is evaluated by the 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:

$$\theta^* = \arg\max_{\theta} \frac{c_i(\theta)}{t_i^{-1}(\theta)}. \tag{13}$$

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

$$s_i = \frac{t_i(\theta^*)}{\sum_{v_i \in \mathcal{L}_{critical}} t_i(\theta^*)} \cdot \mathcal{S}. \tag{14}$$

Algorithm 1: Global-cached Most Cost-effective Critical Path Algorithm

Input: Current function step v_i
Output: Configuration θ_i for v_i
1 Initialize current timestamp τ_i;
2 Get sub-graph G_i starting at v_i and its latest finish timestamp $\mathcal{T}_i$ from global cache;
3 SLO for sub-graph G_i: $\mathcal{S}_i \leftarrow \mathcal{T}_i - \tau_i$;
4 **foreach** *function step* $v_m \in V_i$ **do**
5 Determine θ^* by (13);
6 Set weight $t_m(\theta^*)$ for v_m;
7 Identify critical path $\mathcal{L}_i$ of G_i with weights;
8 **foreach** *function step* $v_m \in \mathcal{L}_i$ **do**
9 Calculate sub-SLO s_m for v_m using (14);
10 Determine θ_i for v_i by solving (12);
11 **foreach** *pair* v_j, v_k, *in* $\mathcal{L}_i$ **do**
12 **if** *exist sub-graph* $G^{'} = (V^{'}, E^{'})$ *between* v_j, v_k *where* $V^{'} \cap \mathcal{L}_i = \varnothing$ **then**
13 $\tau_k \leftarrow \mathcal{T}_i - \sum_{n>k}^{|\mathcal{L}_i|} s_n$;
14 $G^{'} = G^{'} \cup v_k$;
15 $\mathcal{T}^{'} \leftarrow \tau_k - s_k$;
16 Store pair $(G^{'}, \mathcal{T}^{'})$ in global cache;
17 Set $G^{'} \leftarrow \mathcal{L}_i \setminus \{v_i\}$;
18 $\mathcal{T}^{'} \leftarrow \mathcal{T}_i$;
19 Store pair $(G^{'}, \mathcal{T}^{'})$ in global cache;
20 **return** θ_i;

As our online 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. As a result, we establish a shared global cache for them to access the necessary data, which can also reduce the amount of local computation. Based on this, we propose Global-cached Most Cost-effective Critical Path Algorithm in Algorithm 1.

To implement the Algorithm 1, 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 deadline of the function step to finish. It can be calculated by adding the timestamp of entering the current function step with its SLO.

Before the first node of the workflow is executed, we first put the entire workflow G and the corresponding latest completion timestamp $\mathcal{T}$ for G into the global cache. For each subsequent function step that is executed, before entering the execution of the function step, we run Algorithm 1 to optimize the resource configuration choice for that function step. The algorithm first obtains the current function step v_i and the current timestamp τ_i (Line 1). Then, it retrieves the sub-graph starting from the current node and the planned latest completion time from the global cache (Line 2). Next, we can calculate the SLO for the current sub-graph (Line 3). After that, we determine the weight of each function step in the sub-graph according to the best cost-effectiveness strategy and identify the critical path (Line 4-7). Subsequently, we can divide the sub-SLO for each function step on the

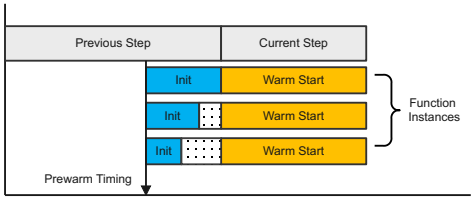


Fig. 10: Diagram of Configuration-Aware function prewarm.

critical path based on their weights (Line 8-9). The optimal configuration choice for the current step is derived from the sub-SLO and the formula (12) (Line 10).

Next, we will identify all sub-graphs connected to the critical path and calculate the latest completion time for each sub-graph (Line 11-15). We store each sub-graph and its latest completion time as a pair in the global cache (Line 16). Finally, we save the remaining critical path as a sub-graph, along with its latest completion time, in the global cache (Line 17-19). In this way, as each function step of the workflow is executed, the scale of the optimization algorithm's optimization is continuously reduced. In the final process, a single node acts as a sub-graph, as nodes (A9,A11) shown in Figure 9.

Theorem 2. *The time complexity of Algorithm 1 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 current sub-graph,* $|\Theta|$ *is the number of configuration combination choices.*

Proof. For Lines 1-3, the time complexity is at most $O(1)$. For Lines 4-6, the time complexity is at most $O(|V|)$. For Lines 7, 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 (12) is at most $O(|\Theta|)$. And for Lines 11-18, 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 (12) 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 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.

5 Configuration-Aware Function Prewarm

In this section, we introduce a solution to solve the cold start problem of the workflow functions with dynamic configuration.

In essence, by leveraging dependencies between different function steps in the workflow, we can prewarm function instances as needed based on the real-time progress of the workflow for future executions, as shown in Figure 10. However, the key to this process lies in the accurate prediction of the function execution process. If the estimation of the workflow's progress is inaccurate, it may lead to incorrect timing for function prewarming, thus failing to effectively mitigate the cold start problem.

In the previous Section 3, we have already introduced a method for performance estimation that can predict the execution times of different function steps under different configurations, including the mapping delay of function instances, in the form of percentiles. Therefore, we can choose the appropriate percentile of performance prediction data based on specific prewarming requirements to accurately determine the optimal timing for function prewarming. If we wish to prewarm functions more conservatively and minimize the occurrence of function cold starts, we can choose larger percentiles, such as p90. Conversely, we can select smaller percentiles, if we want a less conservative approach. With performance estimation in place, we can identify the suitable timing to prewarm functions.

For a function step v_i, its prewarming timing τ_i^p can be determined based on the mapping delay t_i^m of function step v_i under configuration θ, its latest completion time τ_i, and its SLO s_i, by $\tau_i^p = \tau_i - t_i^m - s_i$.

Furthermore, another significant challenge we face in the process of dynamic configuration optimization is the difficulty in accurately predicting the specific instance specifications and their concurrency required for the next function step. To address this issue, we adopt a Configuration-Aware strategy. We first assume that the current function step can be completed as scheduled. Based on this assumption, we subsequently implement Algorithm 1 for the next function step that depends on it, optimize its configuration in advance.

By predicting the resource and parallelism configurations likely to be used in the next function step, we have achieved Configuration-Aware function prewarming. Even if deviations in workflow performance may lead to variations in the final configuration selection, employing this strategy always prewarms some function instances. This prewarmed configuration holds a greater possibility during the optimization of the next function step, as it reduces the mapping delay.

We achieve prewarming of function instances by using a method known as "dummy function invoke" [27]. We assign a special invocation endpoint for each function. When invoked, it triggers the initialization process of the function instance without actually executing the real task processing. Leveraging the cloud platform's function keep-alive policies, concurrent invocations of a certain number of dummy functions can effectively accomplish the prewarming of different numbers of function instances.

6 System Implementation

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

6.1 StepConf Overview

To address a range of issues and challenges in optimizing resource configuration for serverless workflow, this paper introduces StepConf. StepConf combines dynamic optimization of function step parallelism with resource configuration, achieving minimal resource consumption while meeting the SLOs of multi-step serverless workflows. During workflow execution, developers only need to specify

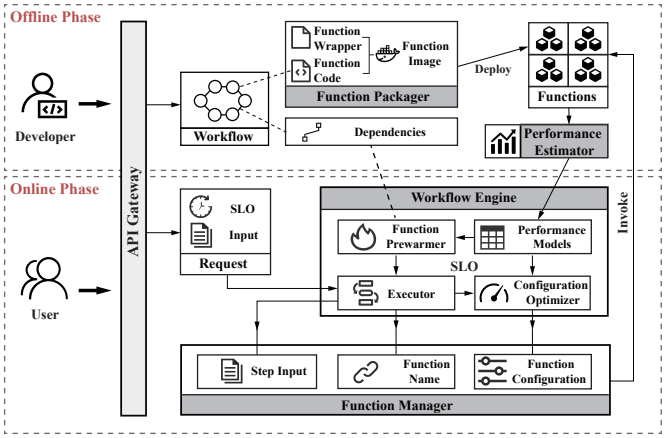


Fig. 11: System architecture overview of StepConf.

the desired SLOs, and StepConf dynamically and in real-time makes decisions to optimize resource allocation before each functional step. Moreover, the Workflow Engine within StepConf employs a Configuration-Aware proactive function prewarm mechanism, significantly reducing the additional overhead caused by cold starts. The system architecture of StepConf, as shown in Figure 11, encompasses both offline and online phases.

During the offline phase, developers need to prepare the workflow definition, which contains the source code of the functions and the dependencies between the function steps. The Function Packager compiles and packages the user-supplied source code along with the Function Wrapper code into function images, which are then deployed. At the same time, the Performance Estimator performs performance profiling of the functions, collecting profile data during function execution and creating performance models for each function.

During the online phase, developers send requests through the API gateway containing the workflow's input and the desired SLO to invoke the workflow. The Workflow Engine, which serves as the core component of StepConf, executes each function step by step based on the dependencies defined in the workflow. The executor tracks the execution status of each function step and initiates execution when the dependencies for the next step are satisfied. Within the Workflow Engine, the Configuration Optimizer plays a central role in StepConf, dynamically optimizing resource allocation based on real-time runtime constraints and function performance models to ensure optimal execution of each function step within the workflow.

In addition, StepConf's workflow engine includes a Function Prewarmer that addresses the challenge of cold starts. By leveraging performance models, it predicts the expected completion time for each function step and warms up the subsequent step accordingly. It also uses the Configuration Optimizer to determine the most likely configuration for the next step and prewarms functions based on that configuration. In the StepConf framework, the Function Manager serves as an intermediary for function execution. It manages the entire function step lifecycle and invokes the functions deployed on the FaaS cluster based on the step input, function name, and function configuration provided by the Workflow Engine.

```
def handler(input_obj):
    # Perform computation
    output_obj = obj_detection(input_obj)
    # Finish computation
    return output_obj
```

Fig. 12: User code example of object detection function step.

These components work together to ensure StepConf effectively uses resources and meets SLOs for serverless workflows while minimizing operational costs. The following sections explain each of these components in more detail.

6.2 Workflow Engine

The workflow engine is used to schedule and sequentially execute each function step based on the dependencies defined in the workflow. Many commercial FaaS platforms have introduced corresponding workflow engines, such as AWS Step Functions [28], Azure Logic Apps [29], etc. However, these workflow engines do not support dynamic resource configuration for workflows, which means they cannot perform real-time resource configuration optimization for each workflow request.

To solve this problem, we implement a workflow engine that focuses on dynamic resource configuration based on Luigi [30]. It not only implements on-demand invocation of function steps according to workflow dependencies but also integrates the Performance Estimator (Section 3) and the Configuration Optimizer (Section 4). These two components are used for the dynamic resource configuration of the workflow.

6.3 Function Packager

Another important component of StepConf is the function packager. It is a program for helping developers to deploy workflows. It is a Python package that provides a convenient API, allowing users to conveniently package their application workflow code into function images and deploy them to the FaaS platform. The function packager only requires the user's application workflow function's handler code and a dependency description file. It then creates the corresponding function Docker image and deploys it to the corresponding cloud platform. The user's provided handler code only needs to follow the template shown in Figure 12, and the function packager will incorporate the Function Wrapper class we designed to override the user's entry function.

6.4 Function Wrapper

The Function Wrapper is a Python class responsible for invoking the user-provided handler function within the function instance using multiprocessing, thereby enabling intra-function parallelism. Before executing each function step, the Configuration Optimizer optimizes the configuration selection for the current step by considering both inter-function parallelism and intra-function parallelism.

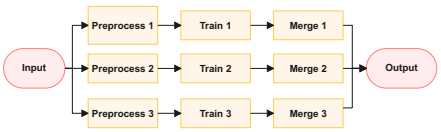


Fig. 13: Machine learning workflow.

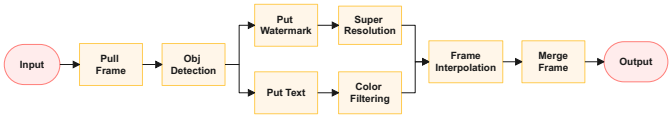


Fig. 14: Video processing workflow.

The workflow engine concurrently invokes the specified number of function instance based on inter-function parallelism. Within each function instance, the Function Wrapper takes over the entire process of data downloading, computation, and data uploading. It invokes the given number of processes based on intra-function parallelism. The Function Wrapper is responsible for splitting the data and mapping it to each process and then waits for all processes to complete before merging the outputs. Additionally, we have rewritten the SDK for accessing storage media in the Function Wrapper using multithreading, enabling parallel data uploading and downloading to improve data transfer efficiency.

In particular, since the current FaaS platform does not support dynamically adjusting the configuration of function resources, how can we dynamically invoke functions with corresponding memory sizes on demand? To address this, we deploy duplicate functions with different memory sizes. We use different versions of the same function to be configured with different memory sizes, using tags like "helloworld:1024MB". In this way, we can select the desired function memory by invoking the function with different URL tags.

6.5 Function Manager

Within the StepConf, the Function Manager plays a central role as middleware and assumes comprehensive control over the execution of function steps. It manages the entire lifecycle of function steps and facilitates parallel processing between functions by invoking a certain number of function instances simultaneously. After the workflow engine provides the step input, function name, and function configuration information, the function manager starts working.

To achieve this, we designed the Function Manager as an internal proxy service within the cluster. During its operation, the Function Manager, guided by the Workflow Engine directives, invokes specific functions that were previously provisioned in the FaaS cluster. Importantly, by using Function Manager, we ensured compatibility not only with AWS Lambda, but also with other platforms, including various open-source and commercial platforms.

7 Experimental Evaluation

In this section, we demonstrate the advantages of the components of StepConf through a series of experiments. Our experimental evaluation aims to answer the following research questions (RQs):

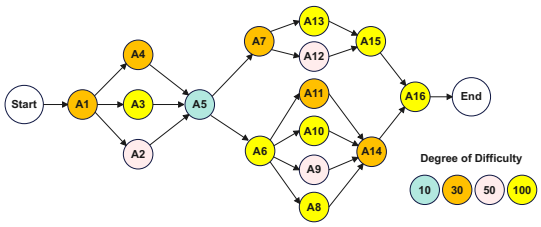


Fig. 15: The synthesized DAG workflow.

- **RQ1:** Can StepConf accurately estimate the performance and cost under different configurations?
- **RQ2:** Can StepConf dynamic resource allocation scheme guarantee the SLOs of workflows?
- **RQ3:** Can StepConf resource allocation optimization algorithm reduce the cost of workflows?
- **RQ4:** Does considering parallelism in StepConf optimization improve the performance and reduce the cost of workflows?
- **RQ5:** Can the StepConf heuristic algorithm approach the theoretical optimum with a small overhead?
- **RQ6:** Can the StepConf function prewarmer reduce the cold start overheads in workflows?

7.1 Experimental Setup

Cluster and Software Information: We build a highly available Kubernetes cluster and installed Knative Serving as the serverless platform, where HAproxy [31] and Keepalived [32] are installed. The cluster comprises 3 master nodes and 6 worker nodes. Each worker node was allocated 16 cores and 32GB of memory, while each master node was allocated 8 cores and 16GB of memory. For data exchange between functions, we utilized an AWS S3 storage bucket located in the nearest region to our experimental environment, ensuring ample public network bandwidth over 1Gbps and low storage access latency to replicate the resource conditions of mainstream cloud applications [33], [34]. Table 2 provides detailed information on the hardware facilities and software versions used in our experiments.

TABLE 2: Experimental environment configurations.

Name	Configuration
CPU	Dual AMD EPYC 7R12 Processor with 96 cores 192 threads (@2.5GHz)
Memory	256GB DDR4 2133 MHz with 16 channels
Disk	Intel P4501 4TB
Network	External network bandwidth over 1Gbps
OS	Ubuntu 20.04 LTS with kernel 5.13.0-51-generic
Kubernetes	v1.27.1
Containerd	v1.6.18
Knative	knative-serving v1.7
S3 Bucket Region	ap-east-1 (Hong Kong)

Workflow Applications: We select three different workflows implemented in Python 3.8 to evaluate the performance of StepConf as follows:

- **Machine Learning Workflow (ML):** Machine learning is a typical use case of serverless workflow. We adopt the ML workflow introduced in [23]. It primarily consists of three function steps, including Preprocess (dimensionality reduction of raw data),

Train (training decision trees), and Merge (merge the results). Our workflow includes three independent ML pipelines, forming the workflow as shown in Figure 13.

- **Video Processing Workflow (VP):** Video processing requires a significant amount of computational resources and is well-suited for leveraging the parallelism of serverless functions to accelerate the processing. Inspired by [4], we establish a complex video workflow. It includes 8 different function steps: pull frame, object detection, watermarking, super-resolution, color filtering, text addition, frame interpolation, and merge frame. The pull frame step decodes the video input into a set of frames, and then each subsequent function step processes this set of frames. The final function step is merge frame to encode the video, as shown in Figure 14.
- **DAG Workflow (DAG):** To thoroughly evaluate the performance of StepConf in a complex workflow environment, we design a complex synthetic workflow application consisting of 16 steps, following [24]. As shown in Figure 15, we select CPU compute-intensive benchmark applications (is-prime) with four different computational complexities, covering a range of upload and download data transmission volumes. Different colors in the figure represent function of varying difficulty. In these 16 steps, we randomly assign tasks of different difficulties. Although this workflow does not have a specific practical use, this method allows us to simulate complex situations that may be encountered in the real world, even though these situations are difficult to fully replicate in a single experiment.

We refer to [17] to define the cost expenses of the Knative FaaS platform. For every single function, we configure different specifications based on the ratio of 1 vCPU to 2048MB of memory.

Baselines: Based on our analysis of recent works outlined in Section 2, we select the following baselines for comparison with StepConf:

- **Vanilla [26]:** Static workflow configuration optimization without considering inter-function or intra-function parallelism, which is based on critical path method.
- **Oracle:** Employs an brute-force algorithm to find the optimal solution for static workflow configuration optimization, jointly optimizing memory size and both inter-function and intra-function parallelism, the same as StepConf.
- **Orion [23]:** Static workflow configuration without considering inter-function parallelism.
- **COSE [22]:** Dynamic workflow configuration without considering inter-function or intra-function parallelism.

We aim to find out the influence of considering both inter and intra-function parallelism and we design baselines as follows:

- **StepConf-Inter:** StepConf's dynamic workflow configuration without considering intra-function parallelism but consider inter-function parallelism.
- **StepConf-Intra:** StepConf's dynamic workflow configuration without considering inter-function parallelism but considering intra-function parallelism.

Table 3 describes the characteristics of each baseline.

TABLE 3: Comparison of StepConf with Baselines

Baselines	Strategy	Inter	Intra
Vanilla	Static (Offline)	×	×
Oracle	Static (Offline)	√	√
Orion	Static (Offline)	×	√
COSE	Dynamic	×	×
StepConf	Dynamic	√	√
StepConf-Inter	Dynamic	√	×
StepConf-Intra	Dynamic	×	√

7.2 Performance Prediction Accuracy

In this section, we evaluate the performance estimator of StepConf through experiments to answer RQ1 (performance estimation). We aim to examine whether our models can accurately fit the performance of different functions under different memory sizes. We select two benchmark functions from Section 2, representing functions optimized for a single core only and functions optimized for multiple cores. By performing performance profiling tests on the function steps and collecting the computational delays for different task numbers, memory sizes, and parallelization strategies, we then use StepConf's performance estimator to fit the test data.

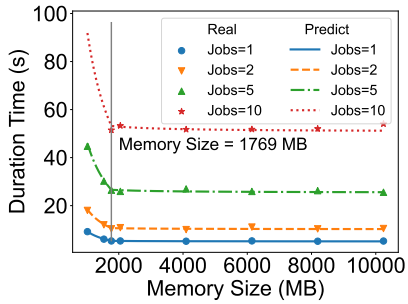
For each function, we conduct profiling for different numbers of jobs using five memory points (512MB, 1024MB, 2048MB, 4192MB, 10240MB). Regarding the varying mapping delay and data transmission, we initially collect actual performance data from a total of 100 recent invocations across different workflows on the cloud platform, to initialize the quantile regression model. As more data is collected by the cloud platform, the quantile model will be able to more accurately reflect the actual performance characteristics and fluctuation trends. The quantile is set as P90 and we repeat the experiment 10 times for average.

Figure 16a describes a function that is optimized for a single thread and does not have intra-function parallelism configured. Therefore, even though the function instances have access to multiple CPU cores, the function cannot utilize them once the memory size exceeds 1769MB. This results in a nearly horizontal line shape in the figures.

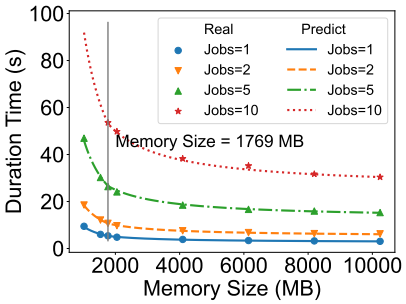
However, when we configure intra-function parallelism for the single-thread optimized function, the function can utilize the multiple core resources, as shown in Figure 16b. Compared to the previous case, enabling intra-function parallelism allows the function to make better use of the allocated resources of function instances, thereby improving performance, given the same memory specifications.

Figure 16c describes a function that is already optimized for multiple cores. From the goodness of fit of the curve in the graph, we can see that the piece-wise fitting model of the performance estimator can accurately fit the performance of the function under different configuration parameters.

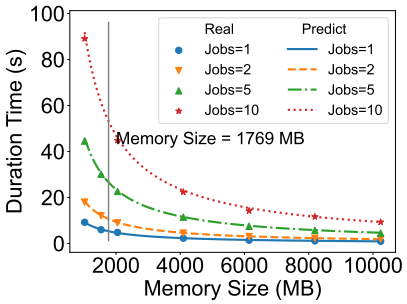
Figure 17 shows the prediction errors for the average end-to-end performance of three different function steps. From the color depth in the graph, it can be observed that,



(a) Single core friendly w/o intra-function parallelism.

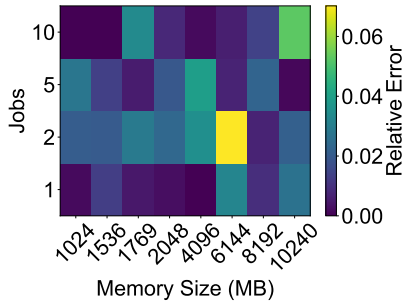


(b) Single core friendly with intra-function parallelism.

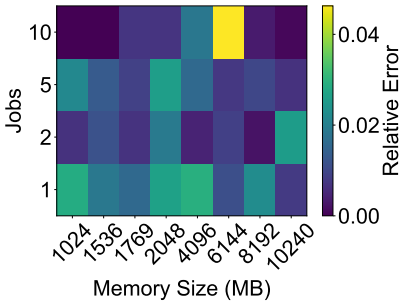


(c) Multi-core friendly with intra-function parallelism.

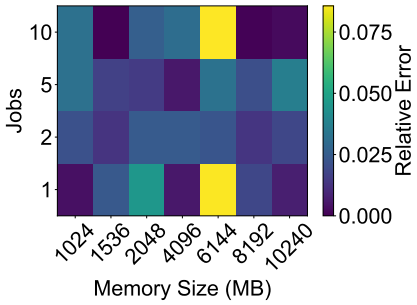
Fig. 16: Comparison of fitting results for functions in different types.



(a) Single core friendly w/o intra-function parallelism.



(b) Single core friendly with intra-function parallelism.



(c) Multi-core friendly with intra-function parallelism.

Fig. 17: Comparison of relative prediction error for different functions in different types.

for different memory sizes and different numbers of tasks, the prediction errors for the functions are mostly within 4%, with the largest error being less than 8%. This accuracy is achieved with relatively low profiling costs, owing to our combined use of piece-wise and quantile models, providing accurate performance estimations.

7.3 Performance Guarantee and Cost Saving

To answer RQ2 (SLO) and RQ3 (Cost Saving), we present experimental results of our workflow executions.

First, we select four different SLOs for each workflow and ran the workflows multiple times, measuring the end-to-end completion time. We collect the performance of the workflow repeated 20 times to see how the SLO fluctuated. As shown in Figure 18, the x-axis represents different SLOs for workflow configurations, and the y-axis represents the normalized workflow end-to-end runtime, which is the actual workflow execution time divided by the SLO. Values closer to 1 indicate that the actual duration time of the workflow is closer to the expected SLO. We can observe that compared to the baseline, StepConf's workflow duration has lower fluctuations and mostly remains below 1. This indicates that StepConf, with its real-time dynamic workflow configuration, better meets the SLOs. It's worth noting that although we can dynamically adjust the workflow configuration in real-time to correct the workflow's execution, the performance fluctuations caused by the last function step cannot be completely eliminated, which is why StepConf cannot completely eliminate performance fluctuations.

Next, we compare the cost of running the workflows under different SLOs. We use Vanilla's performance and cost as the baseline and compare the performance and cost of other solutions. As shown in Figure 19, StepConf can achieve cost savings of up to 40% while improving performance by up to 5.6× compared to the vanilla approach. In comparison to other baselines, StepConf can save costs by 5% to 22.3% while enhancing performance by 1.91× to 3.3×.

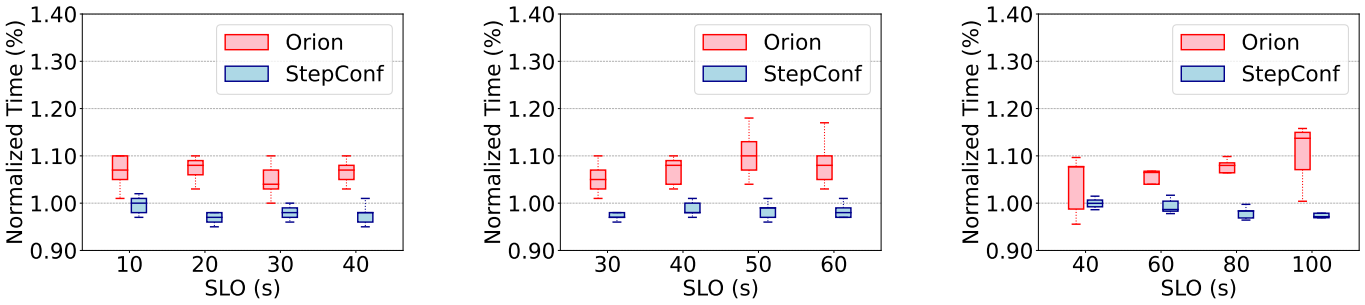
7.4 Impact of Optimizing Parallelism

To answer RQ4 (Parallelism), we investigate the impact of optimizing parallelism in the function steps of workflows and conducted experiments comparing StepConf with partial parallelism optimization.

Figure 20 shows that individually optimizing intra-function parallelism or inter-function parallelism has less impact compared to optimizing both simultaneously. The key point of StepConf is to fully utilize both intra-function and inter-function parallelism, resulting in higher resource utilization efficiency and better performance under the same cost.

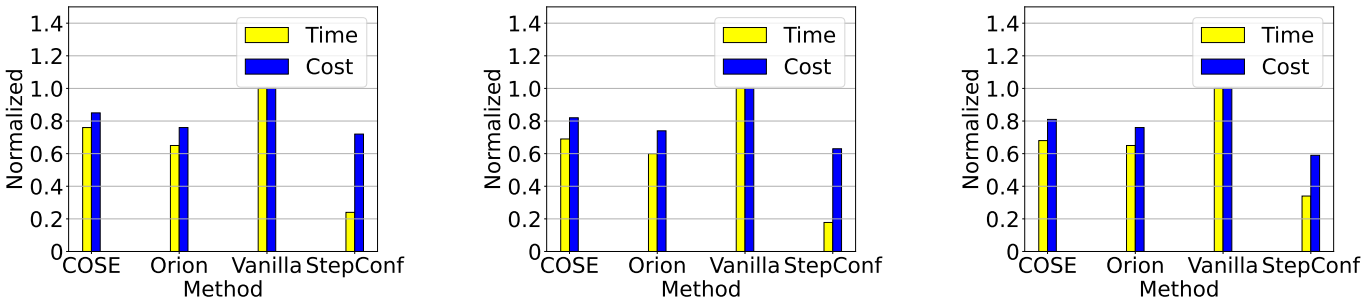
7.5 Algorithm Effectiveness and Overhead

To answer RQ5 (Heuristic), we compare our algorithm with the theoretical optimal configuration namely Oracle. Since online algorithms cannot guide the real-time execution of



(a) Machine Learning. (b) Video Processing. (c) DAG Workflow.

Fig. 18: End to End duration time under different SLOs.



(a) Machine Learning. (b) Video Processing. (c) DAG Workflow.

Fig. 19: Performance and cost compared with different baselines (Lower is the better).

actual workflows, the theoretical optimal solution exists only in offline algorithms. Therefore, we adopt the offline version of the StepConf algorithm, ignoring performance fluctuations in workflow function steps, to make a fair comparison with the oracle algorithm. The oracle algorithm uses a traversal method to find the optimal solution. As shown in Figure 21, our algorithm consistently demonstrates outstanding performance across three workflows, with costs slightly higher than the theoretical optimal solution by less than 5%, respectively 4.1%, 3.5%, and 2.6%.

To demonstrate the efficiency of our scheduling algorithm, we analyzed the delay decomposition in the critical path of the workflow. Figure 22 shows that the delay in configuration decisions accounts for a low percentage, not exceeding 1% of the workflow. In summary, our algorithm performs well in terms of effectiveness and overhead, and it is efficient and feasible in practical applications. For the majority of workflow tasks, the configuration optimization overheads can be considered negligible.

7.6 Function Cold Start Overhead

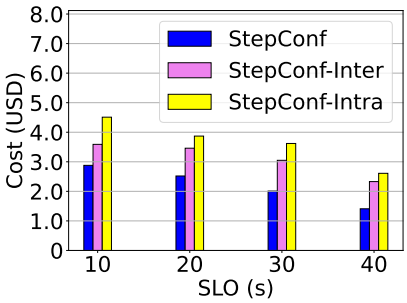
To answer RQ6 (Cold Start), we conduct experiments to evaluate our function prewarming mechanism. According to the definition and description in Section 2, the cold start time of functions falls into the category of mapping delay. Therefore, we compare the proportion of mapping delay in the workflow before and after enabling the function prewarming mechanism, in relation to the overall end-to-end runtime of the workflow.

Figure 23 shows a significant reduction in the average proportion of mapping delay in the workflow. This indicates that the StepConf workflow engine, by incorporating the function prewarming mechanism, reduces cold start overhead, further improving workflow performance and cost savings.

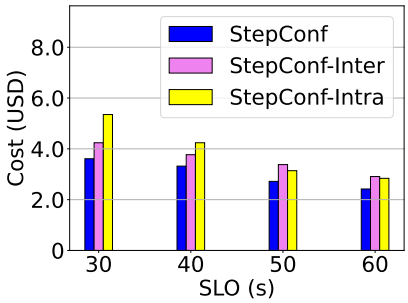
8 Related Work

SLO-Aware Resource Scheduling. There are some existing works that study SLO guarantees [35], [36], [37], [38]. Zhang et al. [36] develop an online, QoS-aware, data-driven cluster manager tailored for interactive cloud microservices. Mao et al. [37] leverage deep reinforcement learning to optimize the scheduling of data processing jobs on distributed compute clusters. Li et al. [38] propose a dynamic VM scaling approach for cost-efficient MapReduce workload management. Romero et al. [39] create an automated, model-agnostic system for distributed inference serving, allowing developers to specify performance and accuracy requirements without choosing specific model variants for each query.

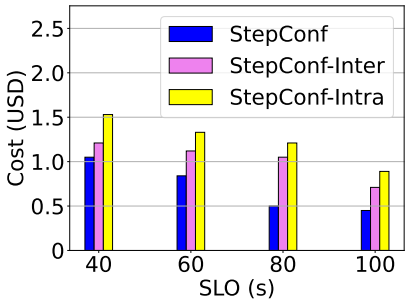
In the serverless domain, Caerus [12] is a task scheduler for serverless analytics, featuring an advanced algorithm that optimizes both execution costs and job completion time. Wukong [13] is a FaaS computing framework for DAG workflows, enhancing data locality and emphasizing decentralized scheduling's benefits in serverless parallel computations, like resource elasticity and cost efficiency. FaaSFlow [14] tackle the inefficiency of central schedulers



(a) Machine Learning.



(b) Video Processing.



(c) DAG Workflow.

Fig. 20: Cost of workflows under different SLOs.

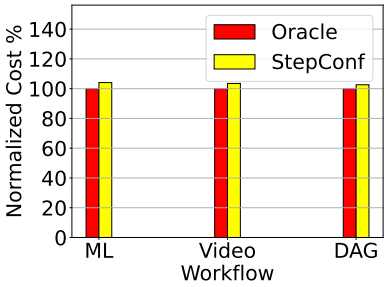


Fig. 21: Avg cost of workflows compared with oracle.

Fig. 22: Breakdown of end-to-end latency with different workflows.

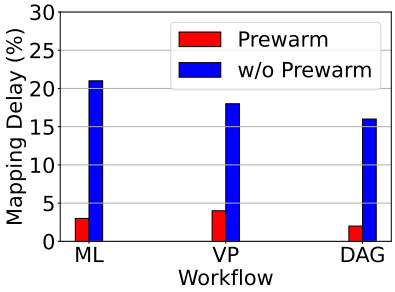


Fig. 23: Reduction of average mapping delay with function prewarmer.

by implementing a worker node-based scheduling strategy, coupled with an adaptive storage repository for faster data transfers between co-located functions. Lastly, Atoll [40] introduce a serverless platform ensuring enhanced latency performance through deadline-aware scheduling, proactive sandbox allocation, and load balancing.

Optimization of Configuration. Optimizing configurations in cloud computing is crucial. Systems like CherryPick [41] and Ernest [42] enhance performance and reduce cost by accurately predicting VM configuration impacts. In the serverless context, this translates to predicting function performance and costs for optimal configuration. Eismann et al. [43] employ mixture density models for predicting function execution times, while Akhtar et al. [22] use statistical learning to optimize function chain configurations. Elgamal et al. [44] investigate cost-effective strategies for function placement and sizing. In addition to previous work [18], [22], [23] discussed in Section 2, Kijak et al. [45] propose a deadline-aware resource scheduling approach for scientific workflows, focusing on cost-efficiency. Lin et al. [24] describes function workflows as non-DAG models, including loops and self-loops. They propose a probabilistic refinement of the critical path algorithm to address the problem of finding the optimal cost under performance constraints and the optimal performance within budget constraints. StepConf addresses this issue through an online approach, jointly optimizing parallelism to significantly save cost while ensuring SLOs.

Function Cold Start. The cold start overhead of serverless functions presents a significant challenge, as highlighted in [2], [46], [47]. Some researchers focus on accelerating the startup time of sandboxes [48], [49], [50], [51], [52], [53], while others aim to reduce the frequency of cold starts [54], [55], [56], [57]. Defuse [58] utilizes historical invocation data to analyze function relationships, prewarming functions to avoid cold starts. Xanadu [59] has also designed a prewarming mechanism for cascading function work chains. Unlike these approaches, StepConf introduces a Configuration-Aware function prewarming mechanism, considering the memory configurations and determining the number of prewarmed container instances based on parallelism, thereby reducing cold start overhead more effectively.

9 Conclusion

We propose StepConf, an SLO-aware dynamic resource configuration framework for serverless function workflows. We develop a heuristic algorithm to dynamically configure each function step, ensuring end-to-end SLOs for the workflow. Furthermore, StepConf utilizes piece-wise fitting models and quantile regression models to accurately estimate the performance of function steps under different configuration parameters. In addition, we design a workflow engine and Function Manager that support various FaaS platforms and reduce cold start overhead through Configuration-Aware function prewarming. Compared to existing strategies, StepConf can improve performance by up to 5.6 $\times$ under the same cost budget and achieve up to a 40% cost reduction while maintaining the same level of performance.

REFERENCES

[1] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," *Research advances in cloud computing*, pp. 1–20, 2017.

[2] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–34, 2022.

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

[4] 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*, 2021, pp. 1–17.

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

[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*, 2017, pp. 363–376.

[7] 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*, 2019, pp. 13–24.

[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*, 2021, pp. 495–514.

[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*, 2019, pp. 475–488.

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

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

[14] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "Faasflow: Enable efficient workflow execution for function-as-a-service," in *Proc. of ACM ASPLOS*, 2022, pp. 782–796.

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

[16] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "Numpywren: Serverless linear algebra," *arXiv preprint arXiv:1810.09679*, 2018.

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

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

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

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

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

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

[23] 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. of USENIX OSDI*, 2022, pp. 303–320.

[24] 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.

[25] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," in *Proc. of IEEE INFOCOM 2022*. IEEE, 2022, pp. 1868–1877.

[26] 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.

[27] "Serverless warmup plugin," https://github.com/juanjoDiaz/serverless-plugin-warmup, 2022.

[28] "Aws step functions: Visual workflows for modern applications," https://aws.amazon.com/step-functions.

[29] "Azure logic apps," https://learn.microsoft.com/zh-cn/azure/logic-apps/logic-apps-overview.

[30] E. Bernhardsson, E. Freider, A. Rouhani *et al.*, "Luigi," https://github.com/spotify/luigi, Spotify, 2022.

[31] "Haproxy load balancer," https://github.com/haproxy/haproxy, 2023.

[32] "keepalived: Loadbalancing and high-availability," https://github.com/acassen/keepalived, 2023.

[33] 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.

[34] F. Wu, Q. Wu, and Y. Tan, "Workflow scheduling in cloud: a survey," *The Journal of Supercomputing*, vol. 71, no. 9, pp. 3373–3418, 2015.

[35] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proc. of ACM EuroSys*, 2019, pp. 1–16.

[36] 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*, 2021, pp. 167–181.

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

[38] 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.

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

[40] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A scalable low-latency serverless platform," in *Proc. of ACM SoCC*, 2021, pp. 138–152.

[41] 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*, 2017, pp. 469–482.

[42] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for {Large-Scale} advanced analytics," in *Proc. of USENIX NSDI*, 2016, pp. 363–378.

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

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

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

[46] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018, 123.

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

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

[49] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "{SOCK}: Rapid task provisioning with serverless-optimized containers," in *Proc. of USENIX ATC*, 2018, pp. 57–70.

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

[51] A. Mohan, H. S. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless." *HotCloud*, vol. 2019, no. 10.5555, pp. 3 357 034–3 357 060, 2019.
[52] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications." in *Proc. of USENIX NSDI*, vol. 20, 2020, pp. 419–434.
[53] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: skip redundant paths to make serverless fast," in *Proc. of ACM Eurosys*, 2020, pp. 1–15.
[54] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. of USENIX ATC*, 2020, pp. 205–218.
[55] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *Proc. of ACM ASPLOS*, 2022, pp. 753–767.
[56] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *Proc. of ACM ASPLOS*, 2021, pp. 386–400.
[57] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *Proc. of IEEE INFOCOM*. IEEE, 2022, pp. 1069–1078.
[58] J. Shen, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu, "Defuse: A dependency-guided function scheduler to mitigate cold starts on faas platforms," in *Proc. of ICDCS*. IEEE, 2021, pp. 194–204.
[59] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proc. of Middleware*, 2020, pp. 356–370.

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

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

Yipei Niu received his B.Eng. degree from Henan University, and Ph.D. degree from Huazhong University of Science and Technology. His research interests include cloud computing, serverless computing, container networking, and FPGA acceleration.

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

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

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