Hindawi
Wireless Communications and Mobile Computing
Volume 2022, Article ID 7862911, 20 pages
https://doi.org/10.1155/2022/7862911

Research Article

SoDa: A Serverless-Oriented Deadline-Aware Workflow Scheduling Engine for IoT Applications in Edge Clouds

Dazhi Li,[1] Jiaang Duan,[2] Yan Yao ,[3] Shiyou Qian ,[2] Jie Zhou,[2] Guangtao Xue ,[2] and Jian Cao[2]

[1]*College of Information, Mechanical and Electoral Engineering, Shanghai Normal University, Shanghai, China*
[2]*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China*
[3]*School of Computer Science and Technology, Qilu University of Technology (Shandong Academy of Sciences), Jinan, China*

Correspondence should be addressed to Yan Yao; yaoyan@qlu.edu.cn and Shiyou Qian; qshiyou@sjtu.edu.cn

Received 1 July 2022; Revised 8 September 2022; Accepted 10 September 2022; Published 7 October 2022

Academic Editor: Mohd Dilshad Ansari

Copyright © 2022 Dazhi Li et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

As a coordination tool, workflow with a large number of interdependent tasks has increasingly become a new paradigm for orchestrating computationally intensive tasks in large-scale and complex Internet of Things (IoT) applications. Serverless computing has also recently been applied to real-world problems at the network edge as well, primarily aimed at event based IoT applications. However, the existing workflow scheduling algorithm based on the virtual machine resource model is inefficient in ensuring the QoS (Quality of Service) of users on the serverless platform. In this paper, we design an elastic workflow scheduling framework in edge clouds called EWSF based on the serverless architecture. In addition, we propose a serverless-oriented deadline-aware workflow scheduling algorithm called SoDa. Furthermore, we implemented the EWSF prototype based on Knative and Kubernetes and integrated SoDa as the scheduling engine. The performance of SoDa has been verified on the experimental platform in comparison with six counterparts. The experiment results show that SoDa adapts to various scheduling environments and achieves better performance in terms of overall makespan and execution success rate. In the case of tight cluster resources, SoDa improves the overall makespan and success rate by 10.4% and 55%, respectively, compared with the second-best algorithm.

1. Introduction

With the increasing number of Internet of Things (IoT) devices, massive amounts of raw data are being generated. As a coordination tool, workflow with a large number of interdependent tasks has increasingly become a new paradigm for orchestrating computationally intensive tasks in large-scale and complex IoT applications. Cloud computing has always been utilized to a great extent in IoT scenarios, offering endless computing capacity and data storage [1]. However, due to the inherently real-time and event-based workload of IoT applications, the latency has proved a challenge for real-time IoT applications in cloud environment.

Edge computing, allowing time-sensitive computations to be executed on compute infrastructure close to the data sources, is a good solution to reduce latency as well as privacy [2–6]. Meanwhile, as a recent paradigm shift, serverless computing would make the IoT application development process even simple for developers. Utilizing serverless, edge computing transforms the previously utilized ship-data-to-code paradigm, which incurred high network latency and transmission costs, to a ship-code-to-data paradigm.

To execute IoT workflows efficiently, enough resources (e.g., CPU and memory) need to be allocated to the tasks of workflows, which leads to many scheduling algorithms [7–9]. Currently, IoT workflow applications are typically deployed in clouds or edge clouds [10]. Users need to set up and configure resources by themselves, which is neither efficient nor convenient.

Serverless architectures like Function-as-a-Service (FaaS) are desirable alternatives for executing workflow applications, turning server-based deployments into service-based

deployments. Generally speaking, a serverless workflow is the orchestration of functions in an application to implement the entire business logic. Compared with traditional architecture, serverless architecture naturally supports the feature of automatic scaling, which can quickly meet the different QoS of various tasks at different layers of the workflow. Therefore, users can focus on their own business logic without having to consider configuring and adjusting resources to meet their QoS. In addition, the serverless architecture adopts a pay-as-you-go cost model, which is cost-effective for users.

Although serverless architectures are easy to use and cost-effective for executing workflows, there are still several issues that hinder their widespread adoption.

First, users are primarily concerned with QoS, such as deadline or execution success rate on time. There are a large number of tasks in a workflow application, and each task has different resource requirements. Heterogeneous resource requirements for tasks can lead to the over- or underprovision problems. Specifically, overprovisioning resources can reduce the efficiency of the cluster, while underprovisioning may cause tasks to fail (e.g., out-of-memory errors). Therefore, the scheduling algorithm needs to comprehensively consider the needs of users to execute workflows efficiently.

Second, although many serverless frameworks are elastic, they do not provide elasticity by themselves, but are actually implemented based on underlying frameworks such as Kubernetes [11]. This elasticity implementation has a gap between user QoS and resource configuration, and it is not adaptive enough to optimize user QoS and resource utilization.

To solve the aforementioned problems, we first design an elastic workflow scheduling framework called EWSF based on the serverless platform. We then propose a serverless-oriented and deadline-aware workflow scheduling algorithm called SoDa, to optimize workflow execution, taking into account deadline and overall makespan. Finally, we implemented EWSF based on Knative and Kubernetes and integrated SoDa as a scheduling engine in EWSF. Furthermore, we present a workflow definition formalism that allows developers to define inherently scalable and generic cloud-native workflows.

We conducted extensive experiments to evaluate the performance of SoDa. We compare SoDa with six counterparts, namely, FWDS [12], ES, PSCP [13], PSWORK [13], WPSCP [13], and WPSWORK [13]. The experiment results show that under the condition of tight resources, the overall makespan of SoDa is 10.4% higher than its counterparts, and the execution success rate has improved by 55%. In addition, SoDa has better autoscaling capabilities, improving resource utilization.

There are four main contributions of this paper:

(i) We designed an edge workflow scheduling framework named EWSF based on the serverless platform

(ii) We proposed a workflow scheduling algorithm called SoDa that takes into account overall makespan and deadline

(iii) We implement the EWSF prototype based on the Knative serverless platform and Kubernetes

(iv) We conduct extensive experiments to evaluate the performance of SoDa on the testbed

The rest of the paper is organized as follows. Section 1 provides the background knowledge. Section 2 discusses the related work. The design of the scheduling framework is detailed in Section 3, and the scheduling algorithm is presented in Section 4. Section 5 describes the implementation of the framework. Section 6 analyzes the experiment results. We conclude the paper in Section 7.

2. Background

In this section, we provide some background knowledge. Note that tasks and functions in this paper are interchangeable.

2.1. Workflow Model. The workflow is widely modeled as a directed acyclic graph (DAG) $G=(V,E)$, where the set of vertices $V=\{v_1,\cdots,v_n\}$ represents the set of n interdependent tasks, and the set of directed edges $E=\{e_{i,j}|\forall v_i, v_j \in V\}$ represents a partial order corresponding to the control and data dependencies among tasks. For instance, edge $e_{i,j} \in E$, v_i is called a direct predecessor of v_j, while v_j is called a direct successor of v_i. Here, we use $\text{pred}(v_i)$ and $\text{succ}(v_i)$ to denote the set of all direct predecessors and successors of task v_i, respectively. A successor task cannot start to execute until all of its direct predecessors have completed running. A task without any predecessor is called an entry task v^{entry}, and a task without any successor is called an end task v^{exit}.

Definition 1. Given a task v_j in workflow G, l_j represents the level of v_j, defined as the length of the longest path from v^{entry} to v_j. The parallelism of v_j is represented by f_j, which indicates the number of occurrences of v_j at the same level in G.

Given a workflow G with m tasks, the data transfer relationship between tasks can be represented by a matrix $\mathbf{D}$ of $m \times m$. As defined in Eq. (1), d_{ij} represents the total amount of data to be transferred from v_i to v_j.

$$\mathbf{D}=\begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1m} \\ d_{21} & d_{22} & \cdots & {}_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ d_{m1} & d_{m2} & \cdots & d_{mm} \end{bmatrix}. \quad (1)$$

If some scheduling algorithms only need one start task and one end task in the workflow, it can be satisfied by adding a dummy start task or dummy end task with empty overhead and zero transmission data.

2.2. Problem Formulation of Workflow Scheduling. In essence, workflow scheduling is to establish a mapping relationship between each task in the workflow and the target

amount of resource in a certain order. Therefore, we need to establish some indicators to reflect the scheduling effect. The performance metrics used in this paper are as follows:

(1) *Makespan and Total Makespan*. Makespan refers to the completion time of a single workflow, which is defined as

$$ms = \max \left\{ \mathrm{ACT}\left(v^{\mathrm{exit}}\right) \right\}, \tag{2}$$

where $\mathrm{ACT}(v^{\mathrm{exit}})$ represents the actual completion time of the task v^{exit}. If there are multiple end tasks, makespan is the largest ACT of these end tasks.

In the multiworkflow scheduling problem, the total makespan is the actual completion time of the last task in all the workflows submitted to the system, which is defined as

$$\mathrm{tms} = \max_{G_i \in \mathbb{G}} \{\mathrm{ACT}(G_i)\}, \tag{3}$$

where $\mathbb{G}$ represents the set of all workflows that needs to be scheduled, and $\mathrm{ACT}(G_i)$ represents the actual completion time of workflow G_i.

Execution success rate. Given a workflow with a deadline, it is important to ensure that the workflow complete by the deadline. The ratio of the number of workflows completed before the deadline to the number of all submitted workflows is called the workflow execution success rate. Let d_i be the deadline of G_i and ms_i represent the makespan of G_i. Then,

$$\mathrm{pass}_i = \begin{cases} 1, & ms_i \le d_i, \\ 0, & ms_i > d_i. \end{cases} \tag{4}$$

Given the workflow set $\mathbb{G}$, the execution success rate is calculated as follows:

$$\mathrm{Success}_{\mathrm{Rate}} = \frac{\sum_{i=1}^{|\mathbb{G}|} \mathrm{pass}_i}{|\mathbb{G}|}. \tag{5}$$

Based on the above definitions, the workflow scheduling problem of optimizing makespan and success rate on the serverless platform is defined as follows:

Definition 2. Given a workflow set $\mathbb{G}$, scheduling is to establish a mapping relationship for each workflow $G_i \in \mathbb{G}$,

$$S_{G_i} = V \longrightarrow F, \tag{6}$$

which means that each task $v \in V$ is assigned to a function $f \in F$ and should satisfy the following constraints: (1) when a task is scheduled, all the predecessor tasks of the task must be executed, and the required data must be ready; (2) the overall makespan must be minimized; and (3) the execution success rate must be maximized.

3. Related Work

Edge cloud is a cloud computing platform built on edge. Service workflow is one of the important difficulties in the edge cloud environment. A dynamic reconfiguration scheme of service workflow in mobile e-commerce environment based on cloud edge was proposed, which was more suitable for edge cloud environment [14]. The task scheduling in edge cloud often has a large number of tasks. For task scheduling in cloud systems, the literature has demonstrated an extensive research body which covers a wide range of task scheduling aspects for optimizing objectives like cost, execution time, energy, reliability, security, and energy [15–17]. Topcuoglu et al. [18] proposed one of the best heuristics scheduling algorithms, the heterogeneous earliest finish-time (HEFT) algorithm to minimize the overall workflow makespan through minimizing the earliest finish time for critical tasks. There exist optimized and extended versions of the HEFT algorithm. However, the technique is not efficient for large-scale workflows.

The concept of serverless architecture was first proposed by Ken Form [19]. In 2014, AWS launched the FaaS platform AWS Lambda [20]. Some research efforts focus on serverless platforms, evaluating or improving the performance of specific serverless platforms. For example, Kuntsevich et al. [21] combine compute-intensive tasks, memory-intensive tasks, and web tasks to perform performance tests on the Apache OpenWhisk platform, measuring the CPU, memory, disk, and latency metrics of the platform. Lin and Glikson [22] propose an optimization method to improve the cold start performance of the Knative platform.

In general, implementing workflow scheduling in a serverless architecture is a challenge because there is no good solution to stateful problems. Malawski et al. [23] review various execution workflows in serverless infrastructures. Kijak [24] outline the challenges of workflow scheduling and design a hybrid serverless deadline-budget workflow scheduling algorithm. The workflow in the above algorithms is limited to a small scale. Jiang et al. [25] propose a workflow execution system in a serverless infrastructure, aiming to execute small- and large-scale scientific workflows while solving the resource underutilization problem. However, the scheduling algorithm batches jobs into Lambda without considering priorities and dependencies, which increases communication costs. To reduce execution costs, Pawlik et al. [26] propose a static heuristic algorithm to scheduling workflow execution based on price, deadline, and budget in FaaS. The algorithm does not satisfy automatic resource management and dynamic scheduling.

Our work is motivated by two aspects. First, scheduling workflows in a serverless architecture no longer needs to consider resource configuration, but should primarily focus on satisfying user QoS. This is different from the goal of most algorithms in traditional architectures. Second, most of the existing serverless platforms mainly rely on the underlying Kubernetes platform to achieve elastic scaling, resulting in a gap between users' QoS and scaling policies. Therefore, in this paper, we focus on implementing a workflow scheduling framework for dynamic workflow scheduling considering

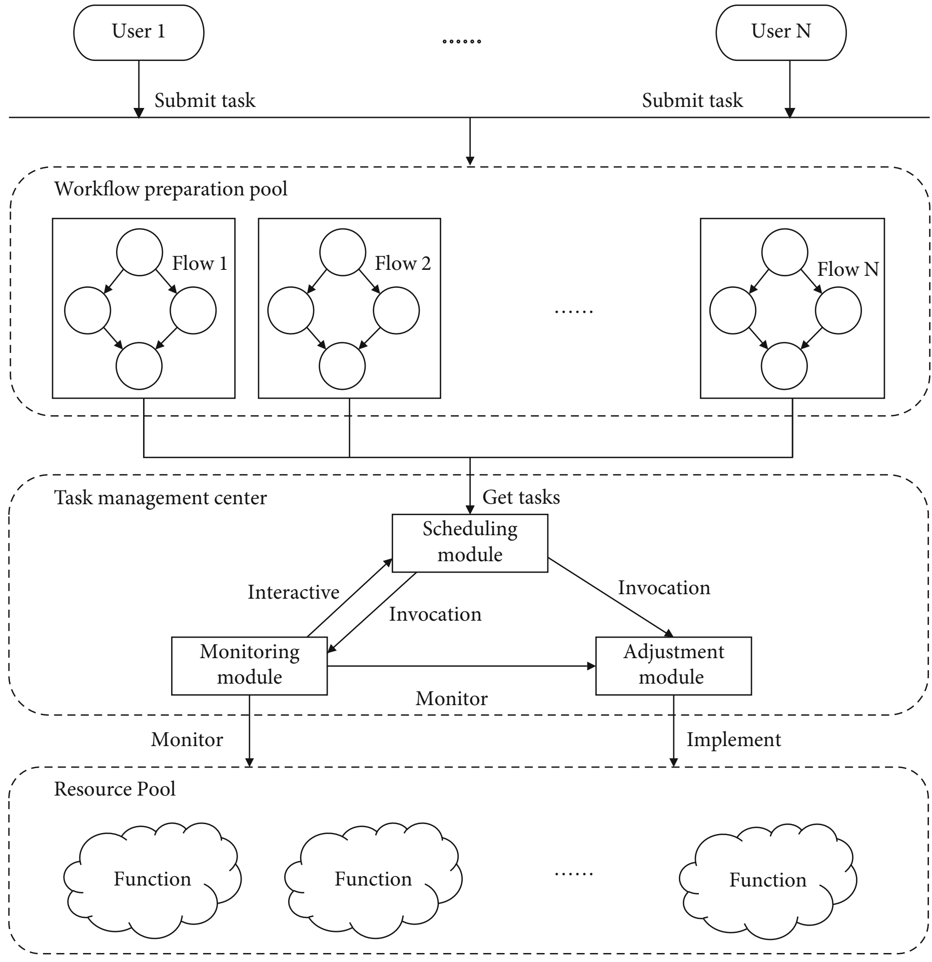


Figure 1: The architecture of EWSF.

deadline and makespan. We use Knative as the serverless platform and Kubernetes as the cluster management system.

4. Edge Workflow Scheduling Framework (EWSF)

4.1. Overview. While designing an edge workflow scheduling framework, we comprehensively consider three aspects. First, since the edge servers are highly dynamic, we need to optimize multiple objectives (e.g., success rate, makespan, deadline, and resource utilization) considering the QoS (Quality of Service) of the client and the resource status of the edge servers. Second, we expect that fine-grained scheduling and adaptive adjustment mechanisms can be implemented in the framework. For example, the number of function instances and the time interval of the scheduling algorithm can be dynamically adjusted based on the QoS of the client and the resource status of the edge servers. Third, the framework should be flexible enough to support different scheduling algorithms for different optimizations in the form of plug-ins.

4.2. The Architecture of EWSF. The architecture of EWSF, which consists of three layers: workflow preparation pool, task management center, and resource pool, is shown in Figure 1. The preparation pool stores workflows and their related information submitted by the client. The task management center mainly monitors the status of running tasks in real time, dynamically adjusts their resource configuration according to the client's QoS and cluster status, and records the task execution results. The resource pool is an abstraction of running environments in the cluster, which realizes the mapping of tasks to resources. The following subsections describe the components contained in these three layers and their interactions.

4.2.1. Preparation Pool of Workflow Instances. IoT workflow applications usually contain many tasks with the same functionality but different inputs. The preparation pool is responsible for converting the workflow into a DAG based on the workflow definition yaml file submitted by the client. In addition, the preparation pool stores the input data of tasks and receives the execution results of tasks. Since different scheduling algorithms follow different workflow data formats, each algorithm has a corresponding preparation pool in EWSF. In addition to the workflow's DAG, other related information is also stored in the preparation pool, such as task completion status and task cost. In the

Require: a workflow G_i, time cycle t
Ensure: Resource configuration of $cpu_{i,j}^{conf}$ and $mem_{i,j}^{conf}$ for each running task $v_j \in G_i$
1: $CPU_{i,j} = \varnothing$, $MEM_{i,j} = \varnothing$ for each running task $v_j \in G_i$
2: $cpu^{\max} = 0$, $mem^{\max} = 0$
3: **For** each running task $v_j \in G_i$ **do**
4: $cpu_{i,j}^{(t)}$ = maximum CPU resource consumed by v_j at time cycle t;
5: $mem_{i,j}^{(t)}$ = maximum memory resource consumed by v_j at time cycle t;
6: Compute ft_{cpu} and ft_{mem} based on Eq. (8);
7: **If** $ft_{cpu} \leq 0.05$ and $ft_{mem} \leq 0.05$ **then**
8: Add $cpu_{i,j}^{(t)}$ and $mem_{i,j}^{(t)}$ to $CPU_{i,j}$ and $MEM_{i,j}$, respectively;
9: $cpu_{i,j}^{conf} = \text{Max}(cpu_{i,j}^{(1)}, \cdots, cpu_{i,j}^{(t)})$, $mem_{i,j}^{conf} = \text{Max}(mem_{i,j}^{(1)}, \cdots, mem_{i,j}^{(t)})$;
10: **Else**
11: **If** $cpu_{i,j}^{(t)} > cpu^{\max}$ **then**
12: $cpu^{\max} = cpu_{i,j}^{(t)}$;
13: **End if**
14: **If** $mem_{i,j}^{(t)} > mem^{\max}$ **then**
15: $mem^{\max} = mem_{i,j}^{(t)}$;
16: **End if**
17: **End if**
18: **If** $CPU_{i,j} = \varnothing$ **then**
19: $cpu_{i,j}^{conf} = cpu^{\max}$, $mem_{i,j}^{conf} = mem^{\max}$;
20: **End if**
21: **End for**

Algorithm 1: Resource configuration of tasks.

Table 1: Knative parameter configuration.

```
newService.Spec.ConfigurationSpec.Template.Annotations = map[string]string{
"Autoscaling.Knative.Dev/metric": "Concurrency",
"Autoscaling.Knative.Dev/window": "10s",
"Autoscaling.Knative.Dev/targetBurstCapacity": "1",
}
containerConcurrency: = int64(1)
newService.Spec.ConfigurationSpec.Template.Spec.ContainerConcurrency = &
containerConcurrency
```

implementation, the workflow preparation pool uses the map object to store all workflows. When the scheduling module needs to fetch tasks, it accesses the map object.

4.2.2. Task Management Center. The functionality of the task management center can be divided into three modules: scheduling, monitoring, and adjustment.

(1) The schedule module implements the workflow scheduling algorithm. The interaction of the schedule module with the other two modules is as follows. Information and resource usage of running tasks can be obtained from the monitoring module. When updating the execution status or adjusting the number of task instances, the corresponding interface of the adjustment module is accessed

Table 2: Parameters of workflow.

Parameter	Yaml key	Category	Necessity
Task sets	tasks	Basic	Yes
Name of workflow	flow	Basic	Yes
Relation set	edges	Basic	Yes
Deadline	deadline	Basic	Yes
Configuration of cluster	cluster	Additional	No
Workflow scheduling pool	pool	Additional	No

(2) The monitoring module is used to collect the resource usage of running tasks in the resource pool and the status of tasks. To support the scheduling algorithm proposed in this paper, the monitoring

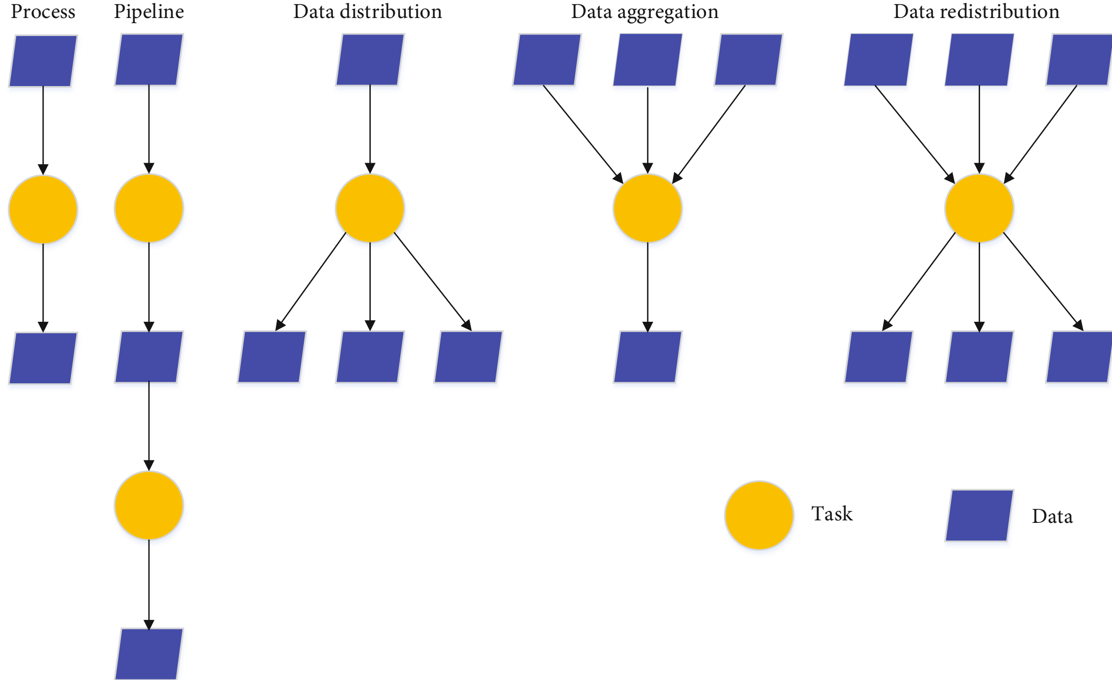


Figure 2: The basic structures of the workflow.

module also incorporates the relevant information of the connection pool into the monitoring scope

(3) The adjustment module is responsible for creating and deleting resources of function instances and controlling the number of requests for a function instance. In addition, the adjustment module is also responsible for the fault tolerance of tasks. Specifically, if a task fails and the number of retries does not exceed the threshold, the adjustment module refactors the requests of the task, and the failure counter is incremented by 1. If the number of failures exceeds the threshold, the task is returned to the schedule module for processing

4.2.3. Resource Pool. The resource pool is an abstraction of the edge servers. Currently, we mainly consider two types of resources, namely, CPU and memory. In the implementation, revision is created through Knative, and resources are allocated to revision and its corresponding pods, so as to realize the mapping process from request to resource.

5. Deadline-Aware Workflow Scheduling Algorithm (SoDa)

The proposed dynamic and flexible scheduling algorithm called SoDa has two main parts: (1) rational configuration of the resources of the function, thereby avoiding the over- and under-provision problem to improve resource efficiency, and (2) dynamical adjustment of the number of function instances according to the QoSs of the client and the status of the cluster, achieving a good trade-off between resource utilization and QoS satisfaction.

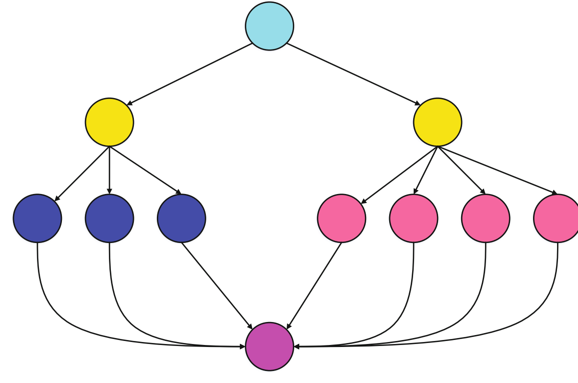
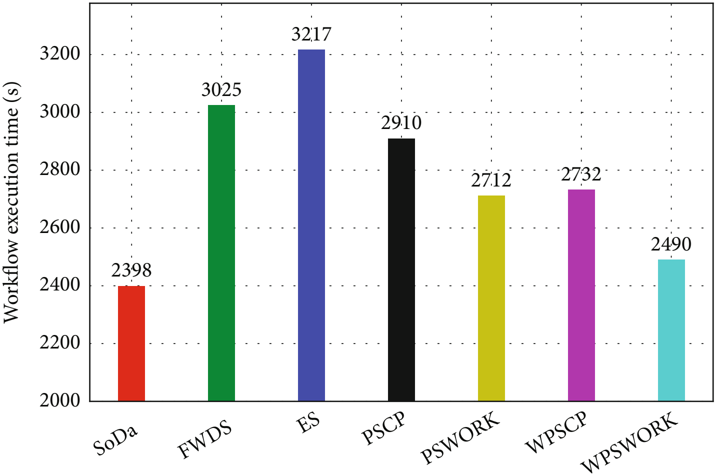


Figure 3: An example structure of highly parallel workflows.

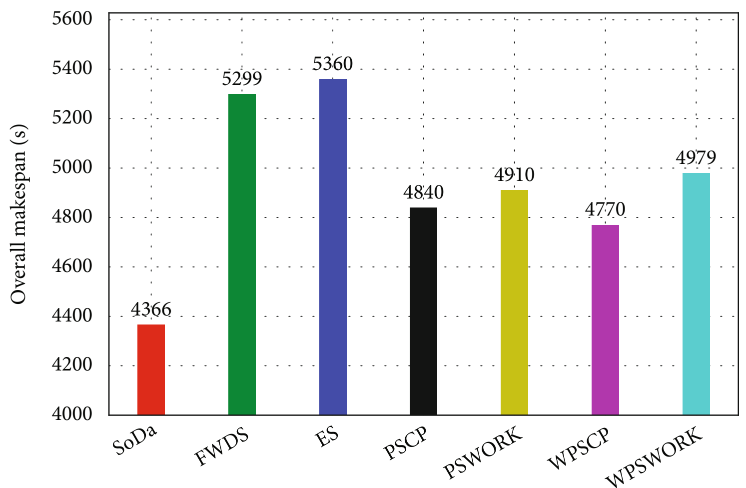
Table 3: Default dataset configuration.

Parameter	Values
Number of workflows	10
Average submission interval	200 s
Workflow deadline coefficient	Loose [1.5,3.0], strict [1.1,1.3]
Number of branches	[1, 5]
Task similarity coefficient	[1, 3]

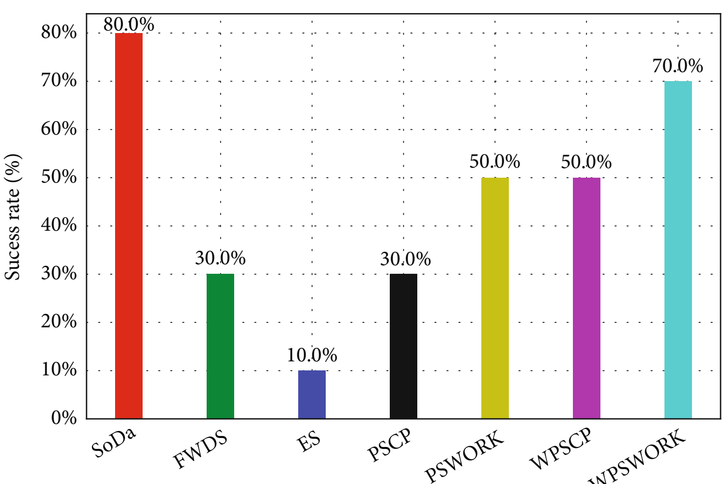
5.1. Rationally Configuring the Resources of Task Instances. We use a feedback mechanism to configure resources for tasks. Each task instance runs in a container. In this paper, we mainly consider two types of resources, namely, CPU and memory. Initially, we set a fixed number of resources for task instances, e.g., 1 vCPU and 4 GB RAM in the implementation. Then, we collect the maximum CPU and



(a) Average execution time

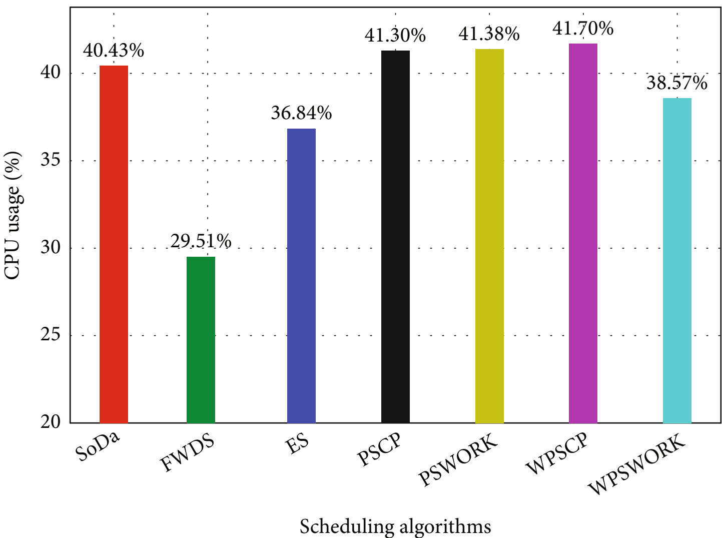


(b) Overall makespan



(c) Success rate of workflows

Figure 4: Continued.



(d) Average CPU usage

FIGURE 4: Results of submitting 10 workflows.

memory usage of task instances per time period. For each running task $v_j \in G_i$, the collected data form two sequences for CPU and memory, respectively. Assuming that there are s time periods, the two sequences can be expressed as

$$\begin{cases} \mathrm{CPU}_{i,j} = \left\{ \mathrm{cpu}_{i,j}^{(1)}, \cdots, \mathrm{cpu}_{i,j}^{(s)} \right\}, \\ \mathrm{MEM}_{i,j} = \left\{ \mathrm{mem}_{i,j}^{(1)}, \cdots, \mathrm{mem}_{i,j}^{(s)} \right\}. \end{cases} \tag{7}$$

Given the time period t and task $v_j \in G_i$, if the following two conditions are met at t:

$$\begin{cases} ft_{\mathrm{cpu}} = \left| \dfrac{\mathrm{cpu}_{i,j}^{(t)} - \mathrm{cpu}_{i,j}^{(t-1)}}{\mathrm{cpu}_{i,j}^{(t)}} \right| \times 100\% \le 5\%, \\ ft_{\mathrm{mem}} = \left| \dfrac{\mathrm{mem}_{i,j}^{(t)} - \mathrm{mem}_{i,j}^{(t-1)}}{\mathrm{mem}_{i,j}^{(t)}} \right| \times 100\% \le 5\%, \end{cases} \tag{8}$$

which means that the resource consumption of v_j stabilizes. Thus, $\mathrm{cpu}_{i,j}^{(t)}$ and $\mathrm{mem}_{i,j}^{(t)}$ are collected by the monitor as resource consumption data.

When scheduling new instances for running task $v_j \in G_i$, the maximum value in $\mathrm{CPU}_{i,j}$ is used to configure CPU resources. The same goes for memory resource configuration. If no time period satisfies both conditions in Eq. (8), the respective maximum usage of CPU and memory is used as the task resource configuration. The pseudocode of determining task resource configuration is given in Algorithm 1. So, to start a new function instance of a running task, we set the number of containers to 1. The number of resources allocated to new instances can be determined based on the maximum CPU and memory usage of the task over a fixed interval.

5.2. Adjusting the Number of Task Instances. Because clusters are highly dynamic, it is important to adjust the number of instances for running tasks to improve resource utilization and deadline satisfaction. To do so, given a running task $v_j \in G_i$, we first dynamically determine the number of resources allocated to it based on the available resources in the cluster, v_j's degree of parallelism f_i, and the deadline d_i of G_i. Then, the number of instances of v_j can be determined. The detailed steps are described below.

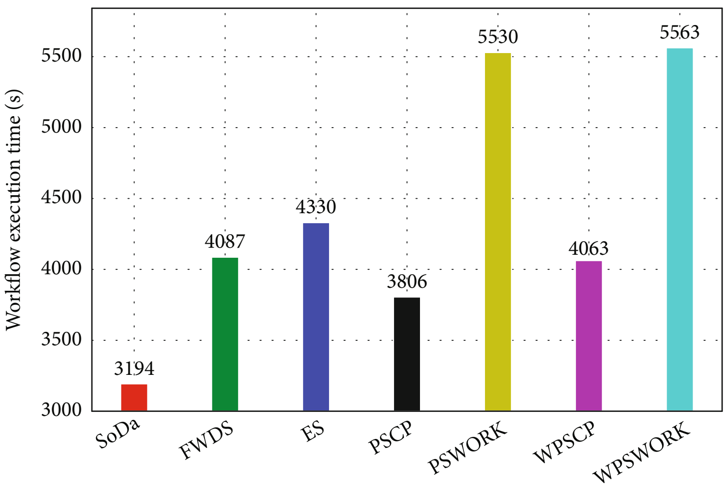
(1) For each workflow G_i with running tasks, let t_i be the start time of G_i, d_i the deadline of G_i, and t_{cur} the current time, the deadline urgency ΔG_i of G_i is defined as

$$\Delta d_i = \frac{t_{cur} - t_i}{d_i}. \tag{9}$$

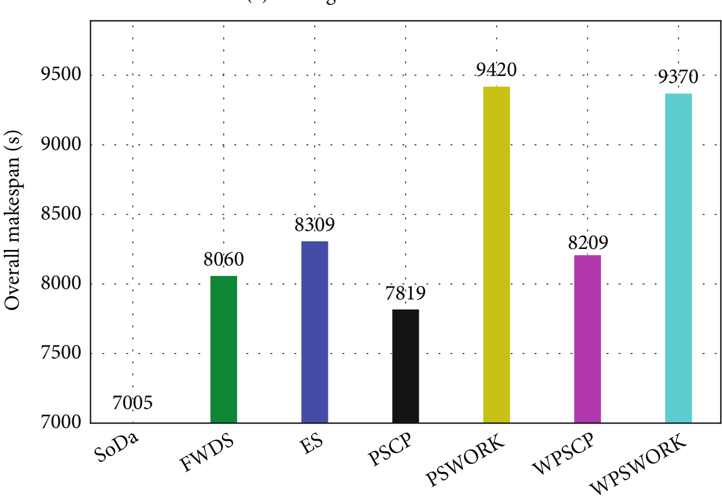
(2) Assuming that there are n workflows with running tasks, and workflow G_k has r_k running tasks, the number of resources for $v_j \in G_i$ is calculated according to the following formula:

$$\begin{cases} \mathrm{cpu}_{i,j} = \mathrm{cpu}^{\mathrm{total}} \times \dfrac{\Delta d_i}{\sum_{k=1}^{n} \Delta d_k} \times \dfrac{f_{i,j}}{\sum_{k=1}^{n} \sum_{l=1}^{r_k} f_{k,l}}, \\ \mathrm{mem}_{i,j} = \mathrm{mem}^{\mathrm{total}} \times \dfrac{\Delta d_i}{\sum_{k=1}^{n} \Delta d_k} \times \dfrac{f_{i,j}}{\sum_{k=1}^{n} \sum_{l=1}^{r_k} f_{k,l}}, \end{cases} \tag{10}$$

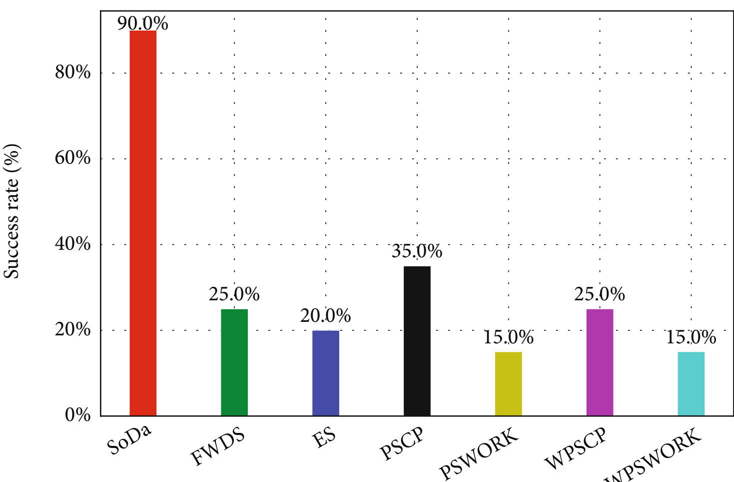
where $\mathrm{cpu}^{\mathrm{total}}$ and $\mathrm{mem}^{\mathrm{total}}$ are the total available resources of CPU and memory, respectively, and $f_{k,l}$ is the degree of parallelism of $v_l \in G_k$. Given the total available resources,



(a) Average execution time

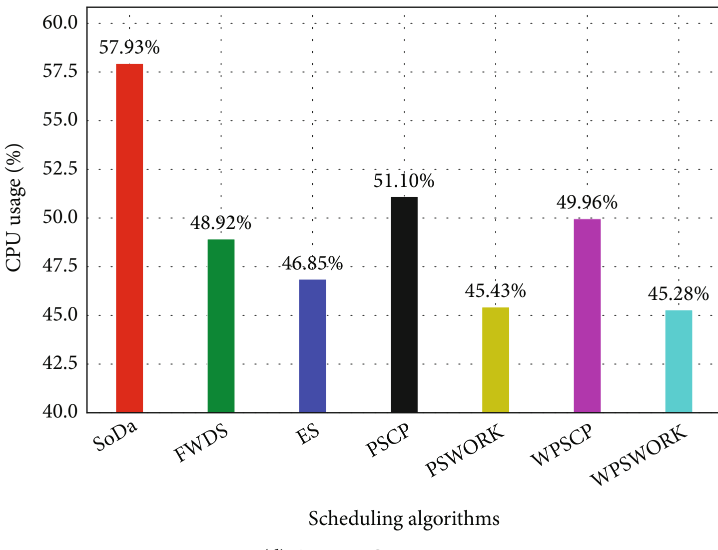


(b) Overall makespan



(c) Success rate of workflows

Figure 5: Continued.



(d) Average CPU usage

Figure 5: Results of submitting 20 workflows.

the number of resources allocated to each running task is proportional to its deadline urgency and degree of parallelism.

(3) According to Algorithm??, the resource configuration $\text{cpu}_{i,j}^{\text{conf}}$ and $\text{mem}_{i,j}^{\text{conf}}$ of $v_j \in G_i$ is determined. Given the number of resources allocated, the number of instances held by $v_j \in G_i$ is calculated as

$$I'_{i,j} = \min\left(\frac{\text{cpu}_{i,j}}{\text{cpu}_{i,j}^{\text{conf}}}, \frac{\text{mem}_{i,j}}{\text{mem}_{i,j}^{\text{conf}}}\right). \quad (11)$$

Based on the current number of instances $I_{i,j}$, if $I_{i,j} < I'_{i,j}$, SoDa starts up $I'_{i,j} - I_{i,j}$ new instances for $v_j \in G_i$; if $I_{i,j} > I'_{i,j}$, $I_{i,j} - I'_{i,j}$ instances are terminated after completion.

5.3. The Scheduling Procedure of SoDa. SoDa is a deadline-aware workflow scheduling algorithm on a serverless platform. The total resources are allocated to the tasks being scheduled according to a certain coefficient which is determined by the deadline of workflows and the degree of parallelism of tasks. The scheduling procedure of SoDa consists of four steps:

(1) The total available resources of CPU and memory are obtained

(2) For each running task, the instance resource configuration is determined using a feedback mechanism, as shown in Algorithm??

(3) According to the deadline of workflows and the degree of parallelism of running tasks, the number of resources allocated to each running task is dynamically adjusted by Eq. (10)

(4) Given the instance resource configuration and the number of allocated resources, the number of instances for each running task is calculated by Eq. (11)

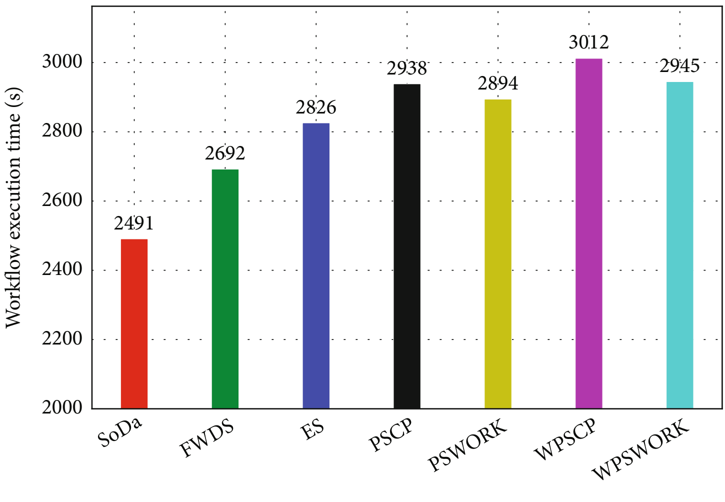
6. EWSF Implementation

6.1. System Implementation. We implemented the EWSF prototype based on Knative and Kubernetes and integrated SoDa as the scheduling engine. Knative is a new serverless platform based on Kubernetes. Knative defines its unique resource organization in the form of Service. Based on the Knative platform, we mainly use the Knative Serving component to implement the proposed EWSF framework. Key features of the Knative Serving component include autoscaling and container routing. In Knative Serving, if clients want to access a service, they do not need to expose that service through traditional Nodeport or Kubernetes Service. Instead, they provide a URL link through which clients can access the corresponding Knative service. At the same time, clients can also access the service through the IP address.

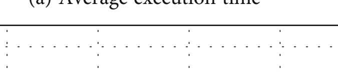
The Knative version used in our implementation is Knative Serving 0.16.0 + Kourier 0.16.0. Since Istio is too large, we use Kourier developed by Google as the network router. In Knative, we configure the parameters listed in Table 1. Specifically, the autoscaling criteria of Knative is set to Concurrency, each pod handles a request, and the autoscaling window is 10 seconds.

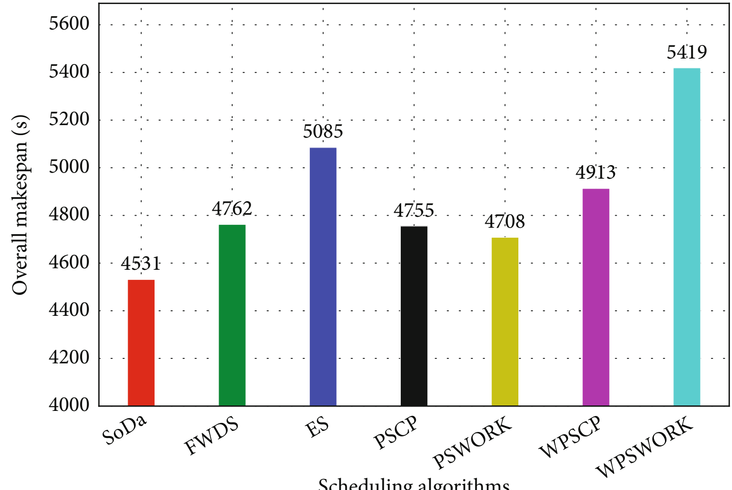
6.2. Encapsulation of Knative Serving

6.2.1. Service Management. Each task of the workflow is abstracted as a separate service. Since a workflow may contain multiple parallel tasks, to improve service efficiency, it

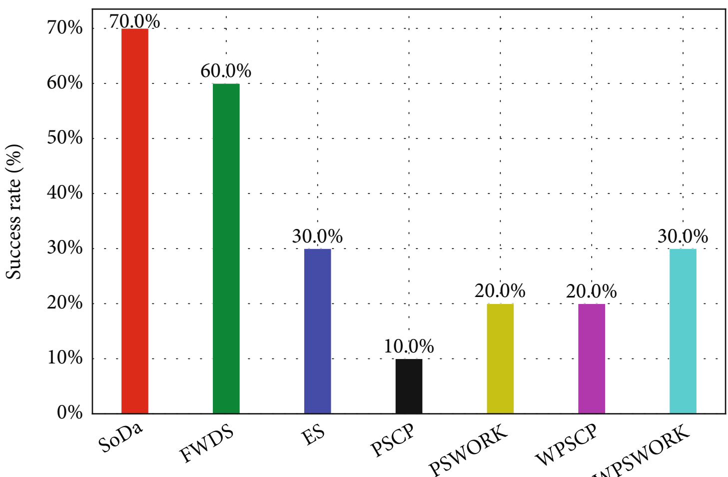


(a) Average execution time



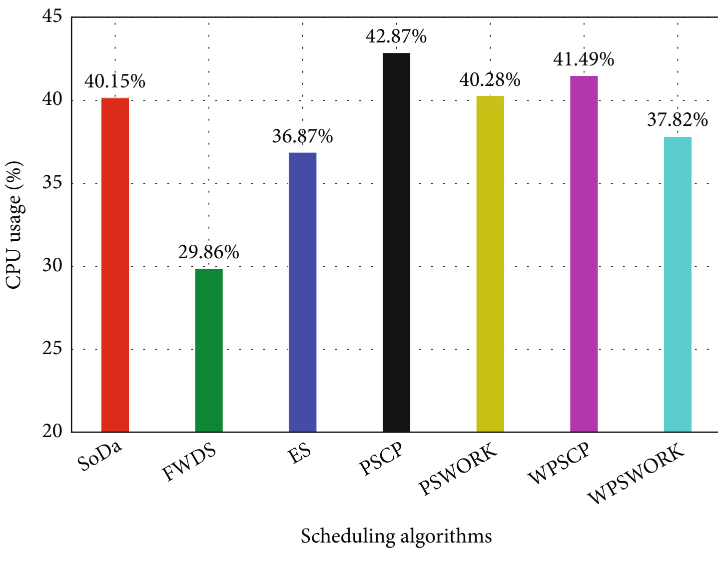


(b) Overall makespan



(c) Success rate of workflows

Figure 6: Continued.



(d) Average CPU usage

Figure 6: Results of 10 workflows with strict deadlines.

is necessary to manage the created services, including operations such as get, list, create, and delete. In our implementation, we use the V1 API provided by Knative Serving to manage services.

6.2.2. Service Configuration. Knative Serving has three autoscaling indicators: concurrency, CPU, and RPS. Concurrency refers to the number of requests that each application can process concurrently at any time; CPU refers to the maximum number of CPUs each application can use (in units of 1 m, i.e., 0.001 CPUs); RPS refers to the number of requests per second that each application can process.

In our implementation, to be compatible with long tasks and short tasks, we choose to use the Concurrency indicator. Knative Serving has a limit on the number of requests that can be processed instantaneously, which is set by the spec. configurationspec.template.

spec.containerconcurrency property of the service. In Knative Serving 0.16.0, the maximum number of transient processing requests is 10 times the containerconcurrency property. In our implementation, we set this property to 4, which means that the system can handle up to 40 transient requests.

6.3. Workflow Definition Schema. We defined a workflow template based on the yaml format with strong expressiveness and a concise format on the Knative platform, allowing users to easily define specific workflows by simply writing the yaml file and packaging the Docker file. In the formal workflow definition, we divide the workflow parameters into the following two categories: basic parameters and additional parameters. Table 2 lists these specific parameters and their descriptions.

7. Experiments

We now present the evaluations of SoDa to demonstrate the efficacy in local cluster running Ubuntu 18.04 with K3s v1.18.6, which consists of one master node and four worker nodes equipped with 32 vCPUs and 64 GB RAM.

7.1. Data Preparation

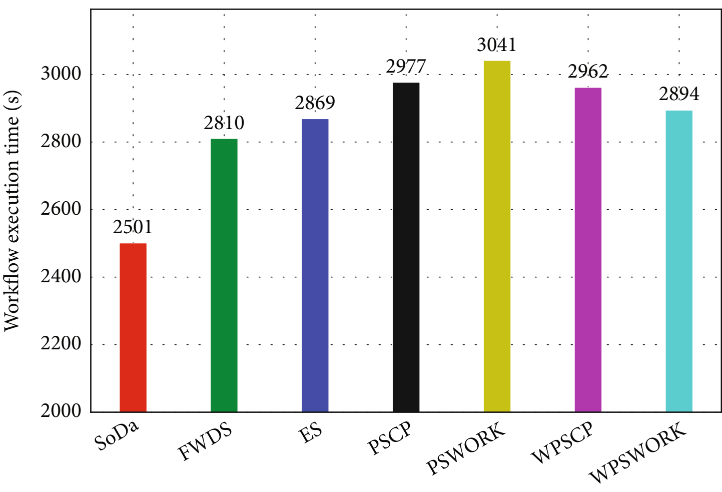
7.1.1. Workflow Instances. We represent a workflow as a directed acyclic graph (DAG) of CyberShake workflow which has several parameters that affect the structure of workflow–number of tasks, DAG width, and critical path length. The basic structures of several commonly used workflows are shown in Figure 2. Moreover, we also propose two additional parameters in order to control the workflow structure–number of branches and task similarity coefficient. The details of the parameters are as follows:

(i) *Number of Tasks*. One of the most obvious characteristics of a workflow is the number of tasks it contains

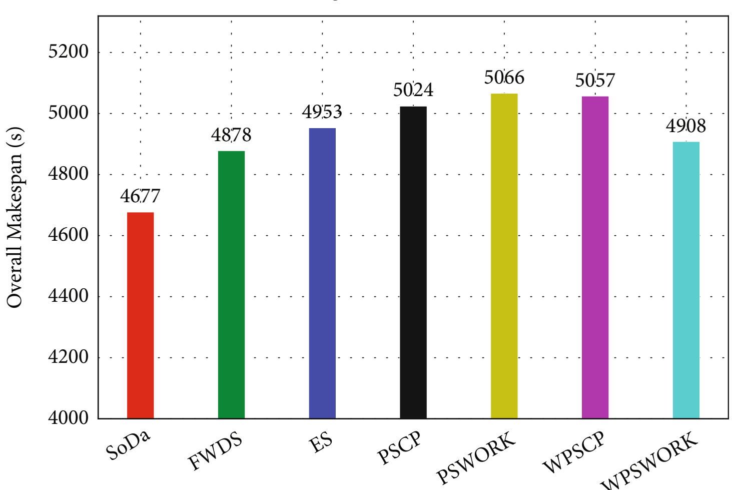
(ii) *DAG Width*. The maximum number of tasks that can be executed simultaneously in the DAG. Relatively small DAGs have small width values. The shape of the entire DAG is like a chain and the task parallelism is low

(iii) *Critical Path Length*. This indicator is used to measure the execution time of the workflow. If the critical path length is short, the execution time of the DAG is relatively short, and vice versa

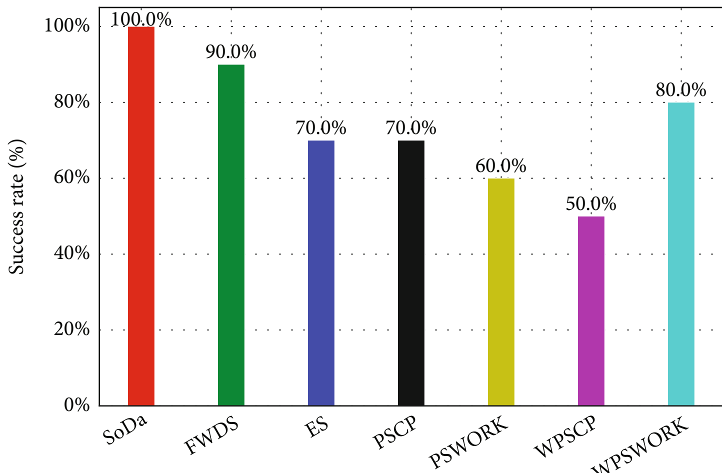
(iv) *Number of Branches*. The number of tasks in the yellow color in Figure 3



(a) Average execution time

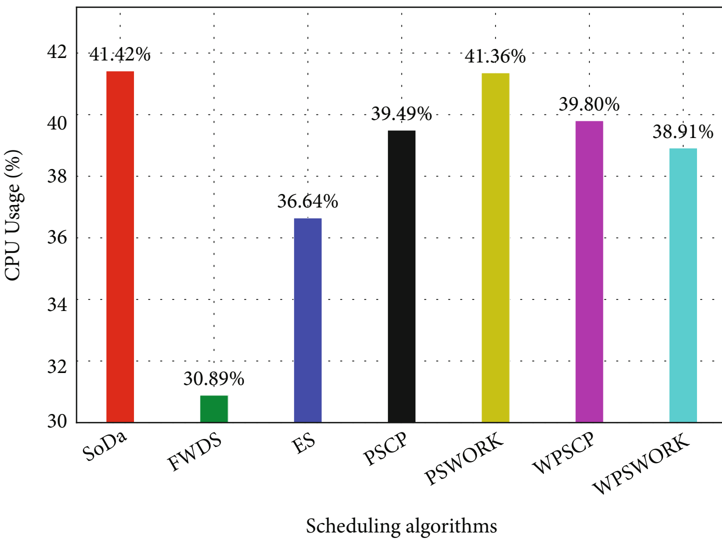


(b) Overall makespan



(c) Success rate of workflows

Figure 7: Continued.



(d) Average CPU usage

FIGURE 7: Results of 10 workflows with loose deadlines.

(v) *Task Similarity Coefficient*. Task types under different branches of the third layer. The lower the task similarity, the more different task types tend to be under different branches

By setting above parameters and referring to the CyberShake workflow, we get workflows with a highly parallel structure, as shown in Figure 3. The first layer is a virtual entry node, which has no practical significance; the yellow nodes on the second layer are called generator; on the third layer, the nodes under each branch are tasks with several connected operators such as matrix calculations stencil calculation adopting the point wise and merge sorting et al.; the fourth layer is the virtual exit node of the entire workflow, having no practical significance.

7.1.2. Default Configuration. For each set of experiments, unless otherwise stated, the default configuration of the workflow dataset is shown in Table 3. Specifically, the deadline for each workflow is set as the product of the minimum execution time among all the tested scheduling algorithms and a given coefficient. The range of the loose-type coefficient is [1.5, 3.0], and the range of the strict-type coefficient is [1.1, 1.3]. The average interval of each submission group is set to 400 seconds. The number of workflows contained in each submission group is randomly selected from 2, 5, and 9. The number of branches of each workflow is 2, and the number of tasks under each branch is 10. Deadline coefficients are randomly generated from the given range. To simulate the real workflow submission process, the submission interval of workflows conforms to the Poisson distribution.

7.2. Metrics. For each set of experiments, we measure the following performance metrics: average workflow execution time, overall makespan, workflow success rate, and average CPU utilization. Of these, the overall makespan and workflow success rate are the optimization objectives of the scheduling algorithm SoDa proposed in this paper.

7.3. Compared Methods. We compare SoDa with the six scheduling algorithms:

(i) *FWDS [12]*. Similar to the priority-based dynamic list scheduling algorithm RANK_HYBD [27]

(ii) *ES*. Evenly allocates total cluster resources to all workflows to be executed

(iii) *PSCP [13]*. A proportional allocation algorithm according to the length of the critical path

(iv) *PSWORK [13]*. An algorithm that allocates resources in proportion to the total number of tasks

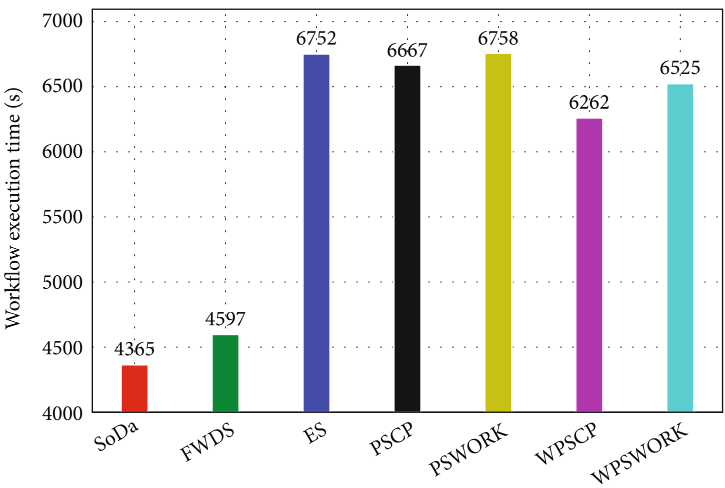
(v) *WPSCP [13]*. An algorithm that allocates resources based on critical path length and coefficients

(vi) *WPSWORK [13]*. An algorithm that allocates resources in proportion to the total number of tasks and coefficients

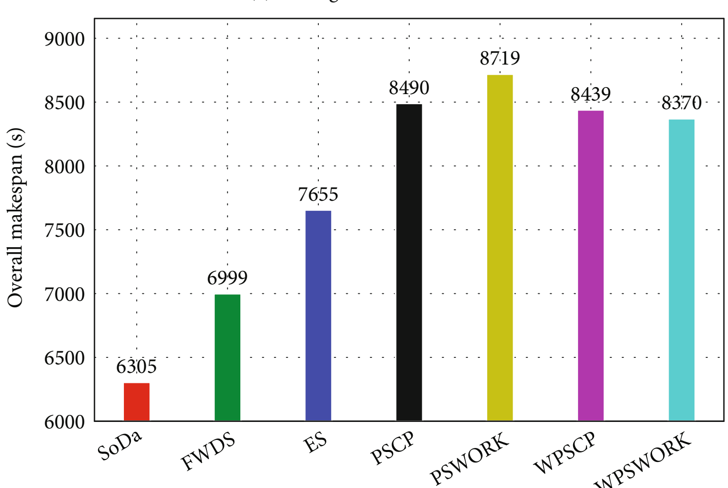
7.4. Experiment Results

7.4.1. Influence of the Number of Workflows. In this experiment, two sets of 10 and 20 workflows are submitted to evaluate the impact of the number of workflows on the four performance metrics. Figures 4 and 5 show the experiment results of submitting 10 and 20 workflows, respectively.

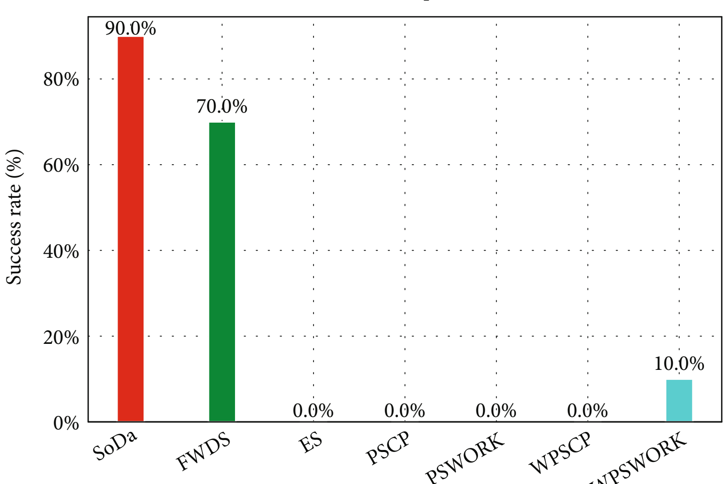
We can see that SoDa outperforms the other algorithms on all four metrics. Figure 4(b) shows that SoDa has advantages over other methods in the overall makespan. Compared with the suboptimal algorithm, SoDa improves the performance of the overall makespan by about 8.5%. Figure 5(b)



(a) Average execution time

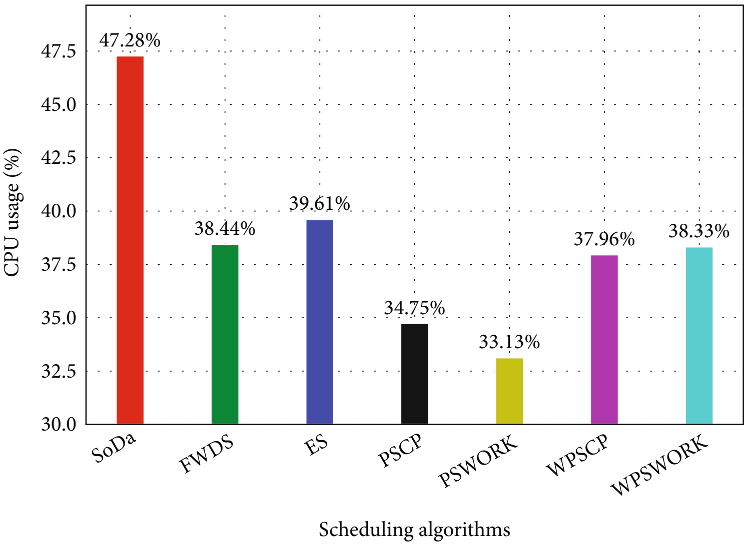


(b) Overall makespan



(c) Success rate of workflows

Figure 8: Continued.



(d) Average CPU usage

Figure 8: Results on the 2-node edge cluster.

shows that compared with the suboptimal algorithm, and SoDa improves by 10.4% when 20 workflows are submitted.

By dynamically adjusting the number of task instances, SoDa can complete workflow execution earlier. In turn, the overall makespan, workflow success rate, and CPU utilization have all improved. This is mainly due to the rational resource configuration of running tasks and dynamical resource adjustment among tasks based on cluster status and the deadline satisfaction. Correspondingly, the success rate of SoDa is nearly 55% higher than that of other algorithms. These results well validate that SoDa achieves the optimization of deadline satisfaction and resource utilization.

7.4.2. Influence of Workflow Deadline Parameters. We evaluate two types of workflows with loose and strict deadline constraints. The experiment results are shown in Figures 6 and 7. From these results, we make three observations. First, SoDa performs best in both cases. The workflow execution time of SoDa is almost the same for both loose and strict workflows. Second, for workflows with tight deadlines, SoDa improves the overall makespan by at least 3.9% over other algorithms due to the fact that the overall demand for resources is relatively large, and each workflow requires a large number of resources to meet the deadline requirements.

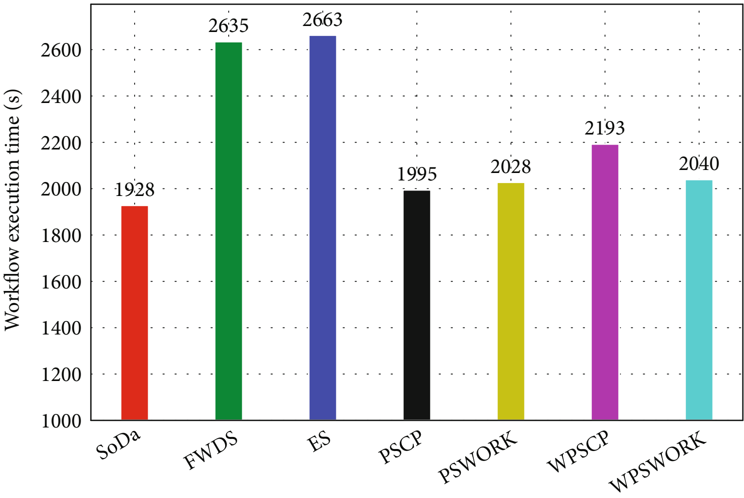
Third, SoDa still has an advantage in the execution success rate for strict deadlines, being at least 10% higher than the other algorithms.

7.4.3. The Influence of Edger Nodes. In this experiment, we evaluate the scheduling algorithms on two edge clusters with 2 and 6 nodes, respectively. Figure 8 shows the experiment results on the 2-nodes edge cluster. Compared with the previous experiment results, the resources become more strained due to the reduced number of nodes. Compared with the suboptimal algorithm FWDS, SoDa achieves a 9.9% improvement in the overall makespan. The average CPU utilization of SoDA is 47.26%, which is about 20% higher than the other algorithms. This is because SoDa uses the autoscaling feature of the serverless architecture to quickly respond to changes in resources allocated to tasks. Therefore, SoDa is expected to have significant advantages over the other algorithms.

The experiment results on the 6-nodes edge cluster are shown in Figure 9. As the number of nodes increases, the edge cluster resources are more relaxed, so the advantages of SoDa are expected to decrease relative to the other algorithms. From Figure 9(d), we can see that SoDa still has the highest average CPU utilization.

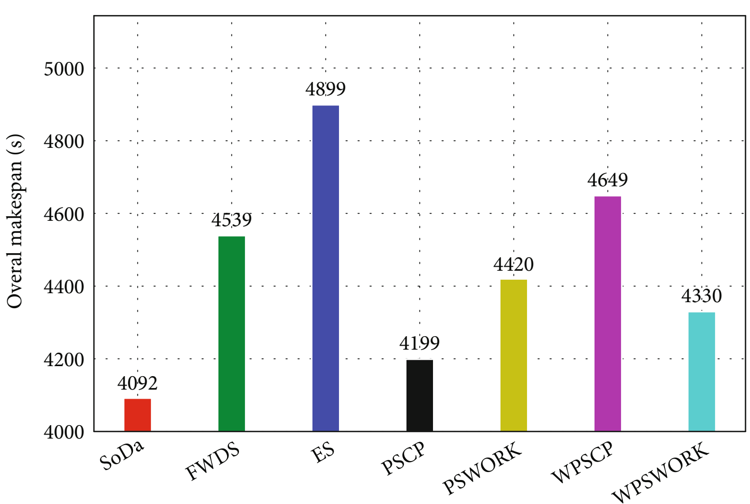
7.4.4. The Evaluation of Auto Scaling Function. In order to evaluate the auto scaling advantage of SoDa, we use two group submission ways to submit workflows: (1) submissions with 3 groups and (2) submissions with 5 groups. To test SoDa without autoscaling, the number of nodes in the cluster is fixed to 6. For testing autoscaling, the number of nodes in the cluster is initially set to 4 and can be expanded to 6. The experimental results are shown in Figures 10 and 11.

When the cluster supports the operation of adding and deleting nodes, SoDa can dynamically adjust the total resources according to the changing workloads. Figures 10 and 11 show the results with relatively light workload submitted by different groups. The resource utilization with auto scaling is higher than that without node scaling, which shows that the method with auto scaling scales nodes when the resource required by cluster load is small and redundant nodes are released. Generally speaking, scaling with nodes has a certain node scheduling overhead compared with scaling without nodes, but correspondingly, it can have higher resource utilization.

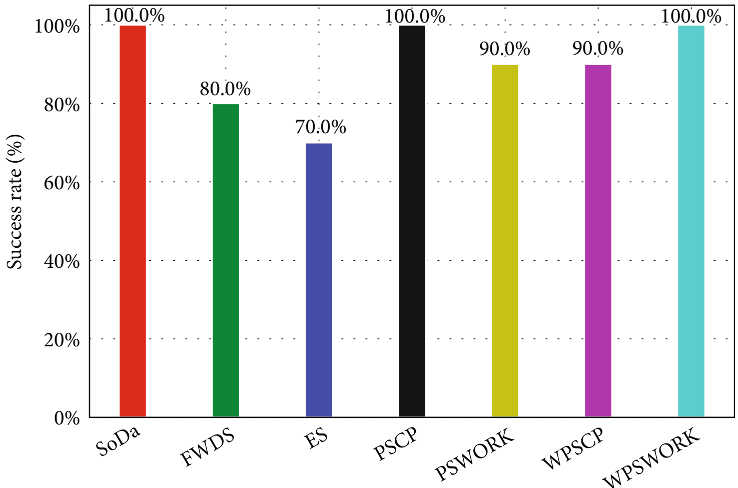


Scheduling algorithms

(a) Average execution time

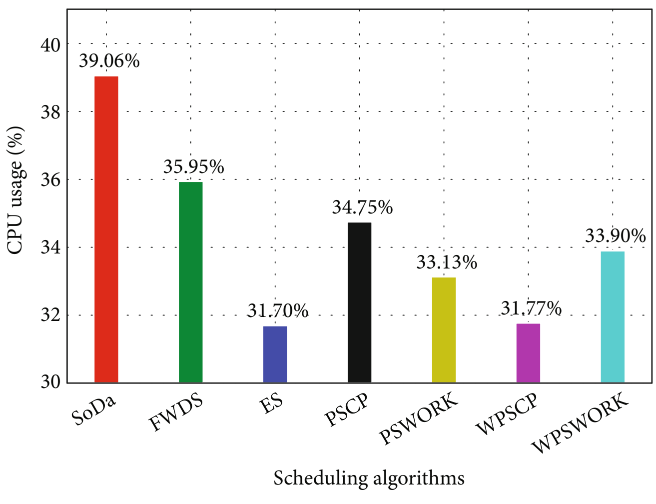


(b) Overall makespan



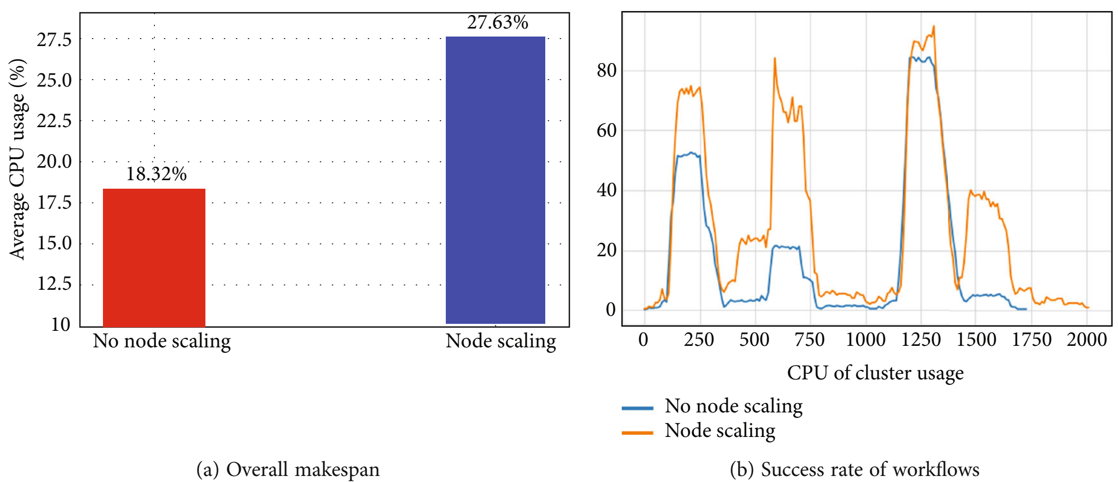
(c) Success rate of workflows

Figure 9: Continued.



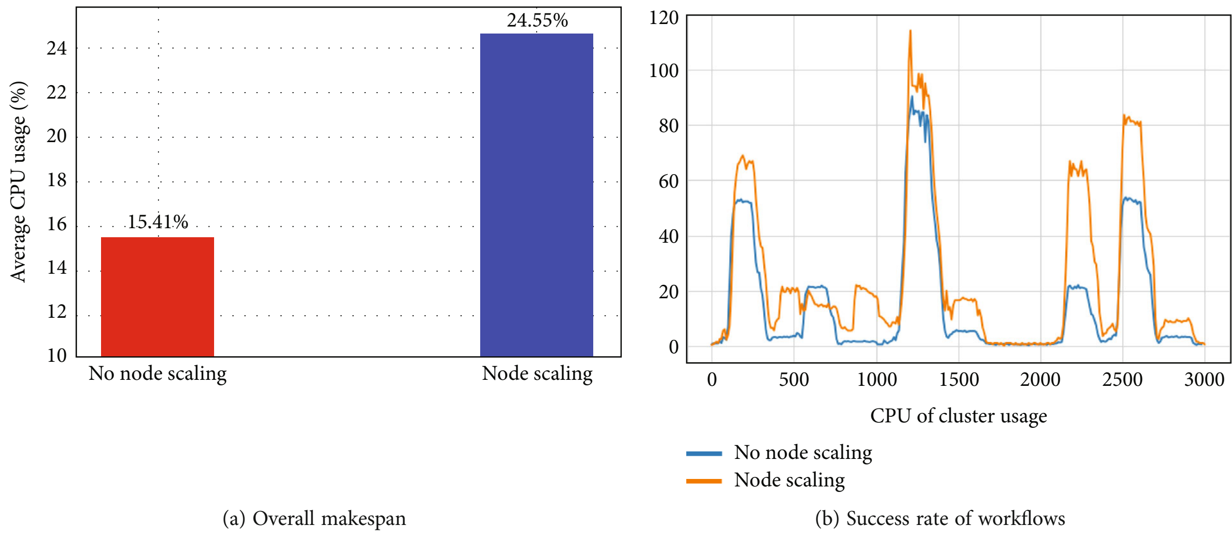
(d) Average CPU usage

Figure 9: Results on the 6-node edge cluster.



(a) Overall makespan (b) Success rate of workflows

Figure 10: Results of submissions with 3 groups.



(a) Overall makespan (b) Success rate of workflows

Figure 11: Results of submissions with 3 groups.

8. Conclusion

We propose an edge workflow scheduling framework (EWSF) in the serverless architecture and implement the framework using Knative and Kubernetes as the technology stack. To address the scalability and interdependence of tasks, EWSF implements scheduling, monitoring, and adjustment functionality. EWSF converts workflow tasks into Knative Serving resources and manages them. In addition, we define a format of workflows, making it easy for clients to specify and create various types of workflows regardless of the operating environments and dependencies. Furthermore, we propose a serverless-oriented and deadline-aware workflow scheduling algorithm called SoDa. By leveraging the autoscaling capabilities of the serverless architecture, SoDa significantly improves workflow execution efficiency in terms of overall makespan and success rate.

Data Availability

No public dataset were used to support this study.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (2019YFB1 and 704400), the National Natural Science Foundation of China (61772334 and 61702151), and the Special Fund for Scientific Instruments of the National Natural Science Foundation of China (61827810).

References

[1] Y. Huang, H. Xu, H. Gao, X. Ma, and W. Hussain, "Ssur: an approach to optimizing virtual machine allocation strategy based on user requirements for cloud data center," *IEEE Transactions on Green Communications and Networking*, vol. 5, no. 2, pp. 670–681, 2021.

[2] P. Gautam, M. D. Ansari, and S. K. Sharma, "Enhanced security for electronic health care information using obfuscation and RSA algorithm in cloud computing," *International Journal of Information Security and Privacy*, vol. 13, no. 1, pp. 59–69, 2019.

[3] L. Ting, M. Khan, A. Sharma, and M. D. Ansari, "A secure framework for iot-based smart climate agriculture system: toward blockchain and edge computing," *Journal of Intelligent Systems*, vol. 31, no. 1, pp. 221–236, 2022.

[4] M. D. Ansari, V. K. Gunjan, and E. Rashid, "On security and data integrity framework for cloud computing using tamper-proofing," in *ICCCE 2020*, A. Kumar and S. Mozar, Eds., vol. 698 of Lecture Notes in Electrical Engineering, pp. 1419–1427, Springer, Singapore, 2021.

[5] J. Zhu, L. Huo, M. D. Ansari, and M. A. Ikbal, "Research on data security detection algorithm in iot based on k-means," *Scalable Computing: Practice and Experience*, vol. 22, no. 2, pp. 149–159, 2021.

[6] M. Ahmed, M. D. Ansari, N. Singh, V. K. Gunjan, B. V. Santhosh Krishna, and M. Khan, "Rating-based recommender system based on textual reviews using iot smart devices," *Mobile Information Systems*, vol. 2022, Article ID 2854741, 18 pages, 2022.

[7] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, 2014.

[8] N. Zhou, D. Qi, X. Wang, Z. Zheng, and W. Lin, "A list scheduling algorithm for heterogeneous systems based on a critical node cost table and pessimistic cost table," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 5, p. 3944, 2017.

[9] H. Chen, X. Zhu, D. Qiu, L. Liu, and Z. Du, "Scheduling for workflows with security-sensitive intermediate data by selective tasks duplication in clouds," *IEEE Transactions on Parallel and distributed systems*, vol. 28, no. 9, pp. 2674–2688, 2017.

[10] A. Yoosefi and H. R. Naji, "A clustering algorithm for communication-aware scheduling of task graphs on multi-core reconfigurable systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2718–2732, 2017.

[11] "Kubernets documentation," https://kubernetes.io/docs/home.

[12] H. Arabnejad and J. Barbosa, "Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems," in *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pp. 633–639, Leganes, Spain, 2012.

[13] T. N'Takpe and F. Suter, "Concurrent scheduling of parallel task graphs on multi-clusters using constrained resource allocations," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–8, Rome, Italy, 2009.

[14] H. Gao, W. Huang, and Y. Duan, "The cloud-edge-based dynamic reconfiguration to service workflow for mobile ecommerce environments," *ACM Transactions on Internet Technology (TOIT)*, vol. 21, no. 1, pp. 1–23, 2021.

[15] M. Adhikari, T. Amgoth, and S. N. Srirama, "A survey on scheduling strategies for workflows in cloud environment and emerging trends," *ACM Computing Surveys*, vol. 52, no. 4, pp. 1–36, 2019.

[16] X. Ma, H. Xu, H. Gao, and M. Bian, "Real-time multiple-workflow scheduling in cloud environments," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4002–4018, 2021.

[17] X. Ma, H. Gao, H. Xu, and M. Bian, "An iot-based task scheduling optimization scheme considering the deadline and cost-aware scientific workflow for cloud computing," *EURASIP Journal on Wireless Communications and Networking*, vol. 2019, no. 1, 2019.

[18] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *EEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[19] K. Fromm, "Why the future of software and apps is serverless," 2012, http://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless.

[20] "What is AWS Lambda?," 2019, https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

[21] A. Kuntsevich, P. Nasirifard, and H. Jacobsen, "A distributed analysis and benchmarking framework for apache openwhisk

serverless platform," in *Proceedings of the 19th International Middleware Conference (Posters)*, pp. 3-4, New York, NY, USA, 2018.

[22] P. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: a pool-based approach," 2019, https://arxiv.org/abs/1903.12221.

[23] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions," *Future Generation Computer Systems*, vol. 110, pp. 502–514, 2020.

[24] J. Kijak, P. Martyna, M. Pawlik, B. Balis, and M. Malawski, "Challenges for scheduling scientific workflows on cloud functions," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 460–467, San Francisco, CA, USA, 2018.

[25] Q. Jiang, Y. C. Lee, and A. Y. Zomaya, "Serverless execution of scientific workflows," in *Service-Oriented Computing. ICSOC 2017*, Lecture Notes in Computer Science, M. Maximilien, A. Vallecillo, J. Wang, and M. Oriol, Eds., pp. 706–721, Springer, Cham, 2017.

[26] M. Pawlik, P. Banach, and M. Malawski, "Adaptation of workflow application scheduling algorithm to serverless infrastructure," in *Euro-Par 2019: Parallel Processing Workshops. Euro-Par 2019*, Lecture Notes in Computer Science, pp. 345–356, Springer, Cham, 2020.

[27] Z. Yu and W. Shi, "A planner-guided scheduling strategy for multiple workflow applications," in *2008 International Conference on Parallel Processing - Workshops*, pp. 1–8, Portland, OR, USA, 2008.